LA PRIMERA REVISTA DE PROGRAMACIÓN EN CASTELLANO



ROGRAMADORES



Revista especializada para usuarios de PC

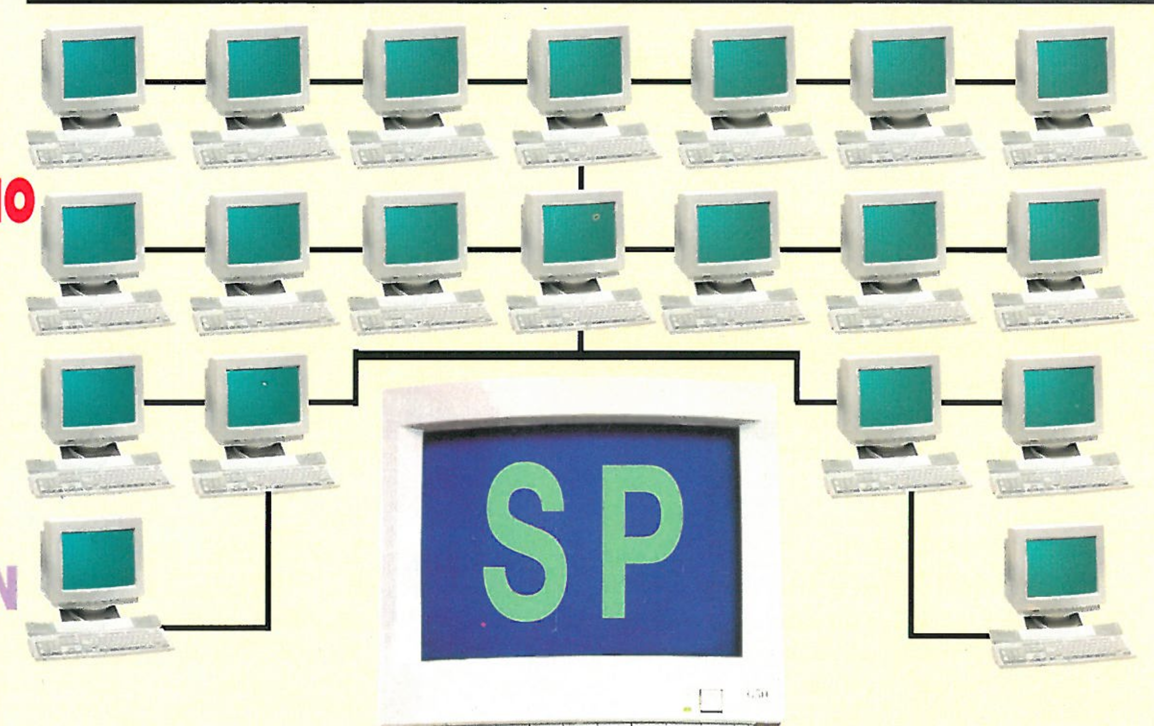
ÑO 3. Nº 31
250 PTAS.

RGENTINA 9'50 $
HILE 3000 $
ORTUGAL 1500$

IIVEL DE USUARIO N LINUX

EMOSCENE: NCLUYENDO BJETOS 3DS

ROGRAMACIÓN ON TCL



GESTIÓN DE PROTOCOLOS NETWARE Novell

Y ADEMÁS
- JavaScript
- Señales en UNIX
- Grandes sistemas
- Inteligencia artificial

URSOS PRÁCTICOS DE:
- Visual Basic 4.0
- Visual C++
- ADA
- Y mucho más



COMMUNICATIONS, S.R.L





Soporte Técnico Borland®

¡ Nuevos ! Cursos a Distancia

Estimado usuario.

Con el fin de facilitarle la formación en las nuevas tecnologías que Borland viene anticipando en sus productos, hemos diseñado para usted nuestros nuevos Cursos a Distancia. De esta forma intentamos ayudarle a solventar las dificultades de disponibilidad de tiempo.

Estos Cursos están basados en los que el Soporte Técnico Borland viene impartiendo tanto en sus instalaciones como a empresas.

¿Qué suponen nuestros Cursos a Distancia?

Semanalmente recibirá una unidad didáctica destinada a la comprensión de los temas, un conjunto de ejemplos para su puesta en práctica, y otro de ejercicios que deberá realizar para el seguimiento personalizado de su curso, y que le serán devueltos corregidos y comentados, junto a la siguiente unidad.

Una vez finalizado el temario, recibirá una prueba de evaluación, que tendrá que cumplimentar. Si el resultado no es totalmente satisfactorio, se repasarán nuevamente los temas requeridos. Por último, se le entregará una acreditación del curso.

Además podrá disfrutar de un año de Soporte Técnico y recibirá una cuenta e-mail y software de conexión a internet, para todo el año, de forma gratuita.

Cursos a Distancia disponibles:

Delphi 2.0
Programación C/C++
Visual dBASE 5.5

Si desea recibir información detallada de los cursos, póngase en contacto con nosotros en:

http: //www.databasedm.es
(también en Infovía)
Tel.: 902 10 20 77
Fax: 902 10 20 66
E-mail: soporte@databasedm.es

Database DM



Número 31
SÓLO PROGRAMADORES es una publicación de Tower Communications

Director Editor
Antonio M. Ferrer Abelló
aferrer@towercom.es
Redactora Jefe
Amelia Noguera
anoguera@towercom.es
Coordinador Técnico
Eduardo Toribio Pazos
Colaboradores
F. de la Villa, Fernando J. Echevarrieta, Pedro Antón, J. M. Martín, L. Martín, J. M. Peco, José C. Remiro, Jorge del Río, César Sánchez, R. Carballo, C. Sánchez, F. Bertran, M. Jesús Recio, Carlos Arias

Jefe de Diseño
Fernando García Santamaría
fegarcia@towercom.es
Maquetación
Clara Francés
Tratamiento de Imagen
María Arce Giménez

Publicidad
Erika de la Riva (Madrid)
Tel.: (91) 661 42 11
Publicidad
Papín Gallardo (Barcelona)
Tel.: (93) 213 42 29

Suscripciones
Isabel Bojo
Tel. (91) 661 42 11 Fax: (91) 661 43 86
suscrip@towercom.es

Filmación
Filma Dos S.L.
Impresión
G. Reunidas
Distribución
SGEL

Director General
Antonio M. Ferrer Abelló
Director Financiero
Francisco García Díaz de Liaño
Director de Producción
Carlos Peropadre
Directora Comercial
Carmina Ferrer
Director de Distribución
Enrique Cabezas García
Asesor Editorial
Mario de Luis

Redacción, Publicidad y Administración
C/ Aragoneses, 7
28108 Pol. Ind. Alcobendas (MADRID)
Telf.: (91) 661 42 11 / Fax: (91) 661 43 86

Depósito legal: M-26827-1994
ISSN: 1134-4792

EDITORIAL



DESTERRAR EL ENSAMBLADOR

Para muchos programadores y personas relacionadas de alguna manera a la informática, el ensamblador es un lenguaje de programación jurásico, algo de museo de Ciencias, que existió en un lejano pasado y tuvo una época de gran esplendor, pero que ya es historia, y cada vez más dado el avance que han tenido los compiladores en los últimos tiempos, ofreciendo mayores facilidades como la programación visual y la orientada a objetos. Todo ello ha supuesto una progresiva marginación del lenguaje Ensamblador, lo cual, me parece cuando menos peligroso.

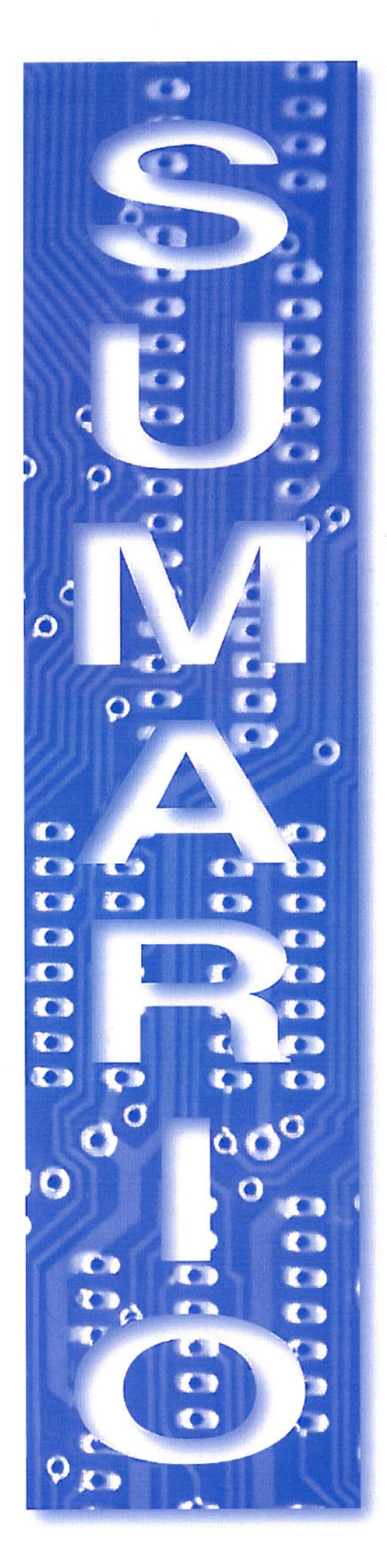
Aunque es lógico que se aprovechen las ventajas de ciertos entornos de programación, cada vez más fáciles de manejar hasta para los neófitos en la materia, que consiguen que personas sin formación sean capaces de desarrollar sus propias aplicaciones, no me parece tan lógico que también se le destierre de los centros de formación de los que serán nuevos programadores. El motivo es que se corre el riesgo de una nueva hornada de desarrolladores de aplicaciones sin un conocimiento claro de por qué se hacen las cosas de una forma o de otra. Creo que está claro que para ser un buen programador se necesita una serie de conocimientos básicos a nivel de hardware, es necesario saber utilizar todos los recursos de la máquina, tener un acceso total al hardware, saber cómo entiende el procesador.

El aprendizaje teórico del lenguaje Ensamblador no nos es fácil para una gran mayoría, y su uso práctico menos aún, no pienso que sea necesario una formación exhaustiva en Ensamblador en la mayoría de los casos, pero sí dedicar algunas horas a este lenguaje. Hacerles saber a los nuevos estudiantes que cualquier facilidad o función que ofrezca un lenguaje de programación está porque puede hacerse en ensamblador, mientras que lo contrario no se cumple siempre.

Está claro que las empresas de hoy en día no exigen ensamblador como requisito, salvo excepciones contadas como alguna empresa de programación que necesite ciertas rutinas para sus aplicaciones. Es más provechoso formarse en C, Visual C, Visual Basic, grandes sistemas o los lenguajes de Autor, pero no se debe marginar y olvidar al rey de todos los lenguajes de programación. Además, aquel que sea un buen programador en Ensamblador se puede decir que es un gran programador, pero el que sea un gran programador en Visual Basic, sólo se puede decir que es un gran programador Visual Basic.

Nosotros, Sólo Programadores, siempre hemos tenido secciones relacionadas con el ensamblador, bien cursos, bien artículos dedicados a la optimización de código, u otras secciones que implementaban sus fuentes en este lenguaje. Creo que una revista de programación no puede olvidar ciertos áreas del campo que cubre.

Eduardo Toribio
etori@ergos.es



31

FE DE ERRATAS
En el número 29 aparece como autor del árticulo sobre comunicaciones en Windows José C. Remiro y debería figurar Carlos Arias.



BETA DE SUITE COMMUNICATOR
Netscape presenta la versión Beta

Entre otros, la nueva Suite integra componentes y funciones de correo e-mail, visualizador de páginas Web y aplicaciones para trabajo en grupo, de forma que los usuarios puedan disponer de las facilidades necesarias a la hora de comunicarse, compartir y acceder a información, tanto en Internet como en intranets. Destacan Netscape LiveWire Pro, entorno de desarrollo y herramienta para la gestión de sitios Web y Visual Café Pro, así como una herramienta visual RAD, diseñada para desarrollar módulos y aplicaciones Java. La Beta presentada ofrece soporte para Windows NT y 95, y en un futuro inmediato estará también disponible para plataformas Unix y Macintosh.

Otros productos presentados por Netscape son Suite Tools, conjunto de herramientas orientadas a intranets, y una guía electrónica con más de cuarenta utilidades, destinadas a la Red y clasificadas en distintas categorías, entre ellas el desarrollo de módulos Java y las utilidades de conectividad a bases de datos.

Para más información:
www.netscape.com

INFORMAT 97 EN PRIMAVERA
22 y 26 de Abril en Barcelona

La Fira de Barcelona ha abierto la convocatoria de Informat 97, el salón de la informática que este año se celebrará entre los días de 22 y 26 del próximo mes de Abril en el recinto Montjuic-2. La Fira de Barcelona aspira a que esta nueva edición responda al importante reto que deben afrontar las ferias de tecnologías informáticas. Se pretende que esta edición sea punto de encuentro entre los diversos profesionales del sector, y que estén representadas las principales firmas del panorama actual. Está previsto para este año la diversificación temática por sectores. Algunos de ellos son la informática corporativa, telecomunicaciones, Internet, CAD para uso doméstico, audio y vídeo o multimedia edición para CD-ROM. El conjunto de la exposición dispondrá de más de 25.000 metros cuadrados.

HISPALINUX 97
El punto de encuentro de usuarios de Linux

Hace ya años que Linux demuestra ser una plataforma seria, viable y fiable para toda clase de aplicaciones. Cada día son más las aplicaciones creadas para él, mayor es su interoperabilidad con otros sistemas y redes, y el número de usuarios y administradores que encuentran a Linux como la plataforma ideal para sus trabajos, estudios y hogares crece.
Los objetivos de HispaLinux 97 son:

- Convertir la feria en el principal foro de discusión y presentación de novedades de habla hispana.
- Creación de un directorio de software creado por diseñadores hispanos que sea de punto de referencia y consulta de interés general.
- Promover y dar apoyo a todas las iniciativas comerciales que den soporte mediante Linux, siempre respetando las diferentes licencias existentes (GPL, BSD, etc).
- Dar publicidad a todo el software libre creado por hispanos en español y que hasta ahora ha encontrado dificultades en su difusión.
- Dar publicidad y difusión a todos los grupos de usuarios dedicados a escribir y traducir documentación y programas al idioma español.

TECNOLOGÍA MMX PARA PENTIUM
Intel prepara los procesadores del siglo XXI

Los procesadores de Intel representarán una referencia clave en el futuro. Así lo cree la propia empresa, que ya cuenta con un calendario en la evolución de estos dispositivos. Durante los próximos quince años se desarrollarán procesadores muy complejos, cuyas principales características serían las siguientes: mil millones de transistores (comparativamente, el Pentium Pro incluye cinco millones y medio), velocidad de reloj de 10 GHZ (atrás quedarán los 200 Mhz actuales), el tamaño bajará de 0,35 micras a 0,1 unidades y los 400 MIPS de 1997 se convertirán en 100.000.

A este respecto, el presidente de la compañía ha declarado que "los cambios sobre nuestra tecnología están destinados a desarrollar microprocesadores más rápidos, baratos y pequeños". El objetivo final parece ser la constitución de un cierto tipo de plataforma informática visual e interactiva. Intel ha reaccionado tras comprobar el alcance a medio plazo de Internet y las aplicaciones multimedia y de comunicaciones, cuya progresión es verdaderamente meteórica.

La división de I+D de Intel entiende que los desarrollos inmediatos en su campo se centrarán principalmente en el sector de la microarquitectura, el diseño de circuitos, la simulación técnica y la tecnología de interconexión, aunque preferentemente se busca lograr la validación de la compatibilidad. Este reto tecnológico resulta muy difícil de superar cuando de diseños complejos se trata o bien de procesamientos mayoritariamente paralelos, ya que la citada compatibilidad seguirá siendo la misma entre los desarrollos actuales y el chip del siglo XXI.

Más en concreto, Intel aseguró que la nueva tecnología MMX para procesadores Pentium podrá mejorar el rendimiento general de los ordenadores personales en torno a un 20%. Se perfeccionará sobre todo lo relacionado con las capacidades multimedia de las aplicaciones que trabajan con vídeo. Asimismo, gracias a MMX los usuarios de Internet e intranets dispondrán también de un mejor rendimiento gráfico y el funcionamiento optimizado de las capacidades de vídeo y audio. Otras ventajas adyacentes son la rebaja en el coste de los sistemas informáticos y un acceso muy sencillo a los entornos de las redes más complejas.

El precio de algunos de los modelos presentados que implementan tecnología MMX ronda las 330.000 ptas.

TECNOLOGÍA DE PROCESO PARALELO
NEC y el INRIA, unidas en un proyecto de investigación

NEC y el Instituto Francés de Ciencias Informáticas, especializado en investigación y desarrollo de tecnología de proceso paralelo, han suscrito un acuerdo para el desarrollo de un proyecto de investigación en el ámbito de la informática de proceso en paralelo. Los objetivos de esta investigación se centran en tres áreas: el diseño e implementación de una memoria virtual compartida de cara a la distribución automática de memoria de forma totalmente transparente al usuario, la evaluación de código producido por Cenju-3 a través de un compilador Fortran y, por último, la optimización de algoritmos paralelos desarrollados por INRIA para que operen en sistemas NEC.

SUN APOYA EL SERVIDOR UNIVERSAL DE INFORMIX
Entorno de soluciones de vídeo digitalizado

La compañía Sun Microsystems ha anunciado su intención de incorporar Informix Universal Server para su uso con el servidor Sun MediaCenter UltraSparc, a partir de este primer trimestre del año. Sun piensa integrar el módulo Video Datablade, con su software de sistemas MediaCenter, lo que permitirá a los clientes de esta solución realizar consultas inteligentes, gestionar datos de vídeo, ampliar bases de datos heredadas y lograr la integración dinámica de vídeo en páginas Web. La nueva solución de Sun estará orientada a todas aquellas compañías que centren su trabajo en base a la gestión de medios digitales. Gracias a este recurso, los usuarios podrán redireccionar rápidamente sus contenidos a la WWW como canal de distribución, a la vez que reducirá costes cuando acorte los ciclos de producción.

Por otra parte, más noticias sobre la compañía establecida en California, Informix, anuncian que varios de los principales puntos de venta en el mundo han seleccionado la arquitectura universal Web de esta compañía para la creación y funcionamiento de iniciativas comerciales en Internet. Entre los nuevos clientes destacan Bass Pro Shops, Bullock and Jones, etc.

La capacidad para acceder a información compleja como audio y vídeo puede facilitar a los posibles clientes la posibilidad de alcanzar pronto el máximo nivel de información. Esto significará una ayuda definitiva a la hora de una posible decisión positiva en la compra de uno u otro artículo.

Para más información:
www.informix.com

SUPERORDENADOR EN ESPAÑA
El CINEMAT adquiere el superordenador más potente del país

El Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas (CIEMAT) se ha apuntado un buen tanto al lograr ponerse a la cabeza del equipamiento informático de alta tecnología con la instalación de un sistema escalable de procesamiento paralelo con memoria distribuida Cray T3E, compuesto de 32 unidades de microproceso sobre chips escalares RISC. El coste total de la operación, que ha llevado a cabo la filial de Silicon Graphics, se eleva a los 400 millones de pesetas, aunque en estos momentos se erige como el ordenador para cálculo científico más potente de cuantos se han utilizado en España. Técnicamente cabe resaltar que es capaz de desarrollar una potencia de 19,2 Gigaflops en su configuración actual.

En cuanto al organismo público, sus recursos de supercomputación van más allá, ya que también disponen de un multiprocesador vectorial paralelo Cray J90 de 16 procesadores. Los responsables de la división informática se muestran abiertos a ampliar el nuevo equipo en un futuro cercano. Los principales beneficiarios serán el personal técnico del Consejo Superior de Investigaciones Científicas (CSIC) y profesores y demás personal universitario.

SYBASE EN LA RED
Sybase crea su puerta propia para entrar en Internet

Las bases de datos de Sybase han aparecido en Internet sin traumas y por la puerta grande, tras conseguir dar con un conjunto de aplicaciones que garantizan que sus sistemas de gestión encajen en la red mundial de consultas. La principal dificultad residía en superar el hándicap que suponen las plataformas y estándares que facilitan las visitas de los navegantes. Pues bien, ahora ya no es necesario pasar por este trago. Sybase recurrió a lo mejor de su catálogo, es decir, lo que ellos definen como productos abiertos, de distribución generalizada, prestaciones muy aceptadas y soluciones de miras globales. Trabajando con estos mimbres se pudieron obtener los nacientes SQL Server Professional para Windows NT, Starbuck y jdbzCONNECT. El primero de los citados permite una mayor optimización en lo referido a creación y gestión de aplicaciones entre cliente y servidor, además de web sites con bases de datos en su interior. Se ha simplificado el proceso a la hora de relacionar conceptos gracias a un conjunto especial de herramientas más productivas que las anteriores, y la implantación sobre ellas de las tecnologías adecuadas.

El producto presenta un mecanismo que prepara y adecúa cada site de la súper red Internet, además de otras referencias de utilidad diversa: SQL Modeler si lo que se pretende es lograr una base de datos realmente profesional, SQL Central cada vez que haya que efectuar una administración gráfica de la oferta temática, y finalmente, una manera propia y distintiva de realizar las consultas e informes que se precisen.

La herramienta de desarrollo rápido RAD Java se puede encontrar en fase beta con la denominación comercial de "Starbuck". Su creación viene a suplir las posibles carencias de los desarrolladores tradicionales, que siempre aspiran a una mayor productividad cuando se trata de analizar y aprovechar las aplicaciones y applets de la citada Java. Sybase ya contaba con una cierta experiencia en lo relativo a productos comerciales y recursos técnicos del Powersoft Optima ++. El entorno que han perfeccionado alcanza buenos límites de rentabilidad ya sea para el cliente o el terminal del servidor, indistintamente. Por ejemplo, gracias a la conectividad escalable a bases de datos y desarrollos. Para hacerse una idea, componentes como JavaBeans y ActiveX son dos de los usuales que caracterizan su entorno.

NUEVO PROCESADOR K5-PR166
Los K5 de AMD se multiplican

La familia de procesadores AMD-K5 ya tiene un hijo más: K5-PR166. Su aparición tiene mucho que ver con la necesidad de cubrir un espacio concreto en el mercado de sobremesa. Su rendimiento, sin ir más lejos, supera al de los Pentium de 166 MHzs. Para llegar a este juicio de valor se ha tenido en consideración la base del Coeficiente P, o lo que es lo mismo, el banco de pruebas Ziff-Davis Winstone 97. Este nuevo producto es compatible del todo con el hardware Socket 7, por lo que se abarata el precio del rediseño y desarrollo del sistema. En cuanto se han conocido sus características ha surgido un fabricante, en concreto Acer, que ya cuenta con el procesador en dos de sus nuevos modelos del catálogo AcerEntra.

El coste económico de los AMD-K5-PR166 se eleva a los 167 dólares por cada uno (21.000 pesetas, aproximadamente), si los dispositivos con los que se va a aplicar son más de mil.

SUNSOFT RECURRE A JAVA PARA SU ESTRATEGIA COMERCIAL

Sus desarrollos y su estrategia comercial se basarán en Java

Tras la visita a España de Philippe Lerer, director general de Sunsoft para la zona mediterránea europea desde su nombramiento el pasado mes de julio, ha quedado bien claro que la mayor parte de sus desarrollos están basados en Java y que su apuesta para el futuro es ésa. A diferencia de 1996, en que la compañía se ceñía a las aplicaciones en lenguajes tipo C y C++, la evolución del mercado parece indicar que muy en breve todas las aplicaciones disponibles en las tiendas correrán bajo este lenguaje. Java adquiere así un protagonismo muy evidente y configura la política inmediata de la compañía, que se centrará en exclusividad en la venta mediante canales. Sin embargo, tampoco se olvidarán de los no iniciados en Java, ya que su intención es perfeccionar una multiplataforma que incluya las máquinas y sistemas operativos de Solaris, Unix, Windows NT, navegadores de Microsoft y Netscape, OS 2 ó AIX.

Al desaparecer la venta directa, Sunsoft optará por reforzar sus lazos respecto a distribuidores y vendedores. Con ello se pretende que el cliente no se aturda ante la enorme oferta de productos y que de verdad conozca la validez de cada uno de éstos. Está previsto que se organice un servicio de consultas a particulares para que sus referencias se distingan bien de las de la competencia, aunque de forma automática se les dirigirá a los puntos de comercialización adecuados.

Haciendo un poco de historia, los lectores recordarán que Sunsoft, filial de la multinacional Sun Microsystems, se ha dedicado desde 1994 a trabajar preferentemente con Unix. Ahora ha llegado el momento de ampliar horizontes y dar el salto a otras plataformas como la NT. El optimismo que se respira en la compañía es evidente. Java les ha proporcionado buenos clientes y se hace todo lo posible para conservarlos por encima de cualquier otra consideración. Un ejemplo bien significativo es la reciente firma de un acuerdo con Solstice, uno de los poderes del omnipresente Bill Gates. Nunca antes Microsoft había tenido que recurrir a soluciones ajenas y el grado de colaboración les ha emparentado felizmente con la major, a la que sin embargo discuten su estrategia Activist. La planificación, dirigida a grandes redes, se centra sólo en el mercado de los Pcs. Frente a ello, Sunsoft prefiere pensar en sus clientes de Intranet como posibles usuarios de otras posibilidades, desde Ncs a juegos, televisiones y otros servicios de futura implantación. Las autopistas de la información son la clave de un crecimiento económico que les permitirá dar cobertura a estos futuros compradores. Aunque antes habrá que ajustar ventas, ya que el coste anual de un sólo PC asciende a una media de 12-15 dólares por empresa y se busca que el NC lo reduzca a 2.5-5 $. Lerer puntualiza que "en eso se basa el éxito de las redes de comunicación y la ventaja que supone Java es que es compatible con cualquier modelo informático del momento, permitiendo ahorrar mucho dinero. Nosotros, en una palabra, tenemos las herramientas para permitir a una compañía sobrevivir en el mundo de la información".

VISUAL BASIC 5.0, PROFESIONAL EDITION

Nueva versión con un rendimiento superior al 2000 por ciento

Microsoft ha anunciado la última versión de la herramienta de desarrollo de aplicaciones Microsoft Visual Basic 5.0, Professional Edition, que presenta mejoras importantes de productividad y rendimiento, basadas en las prestaciones avanzadas del producto, como compilación de código nativo, acceso a bases de datos a alta velocidad y un entorno de desarrollo mejorado.

Además permite el desarrollo de componentes de las tecnologías ActiveX para la creación de aplicaciones orientadas a Internet, Intranet y entornos cliente/servidor. En la actualidad un gran número de desarrolladores están creando controles ActiveX, componentes software compactos y reutilizables en una gran gama de productos como Internet Explorer, Office 97, Visual C++ y Visual Fox Pro, utilizando la versión gratuita de Visual Basic 5.0, Control Creation Edition,

Como objetivo clave a la hora de diseñar Visual Basic 5.0 se ha tenido la mejora de la productividad del desarrollador y la facilidad de uso. Para ello se utiliza la tecnología IntelliSence de Microsoft y un nuevo entorno de desarrollo inteligente que incluye muchas mejoras en el paquete de formularios, el editor de código, el explorador de objetos y el depurador. El entorno de desarrollo es compartido por todas las ediciones de Visual Basic.

Para más información acerca de la familia de productos:
http://www.microsoft.com/vbasic/

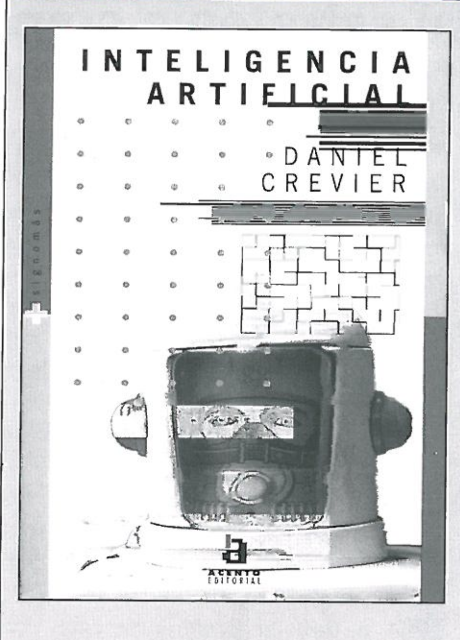
NOVEDADES

VISUAL C++ VISUAL STUDIO 97

La edición profesional del nuevo Microsoft Visual C++ 5.0 presenta en su última versión el desarrollo más actualizado de las herramientas de trabajo de mayor difusión a escala mundial. Los nuevos componentes incluidos en la versión 5.0 aumentan la productividad de la patente.

Se integran en una sola unidad las tecnologías propias de Internet y la relación cliente/servidor. Estos modelos incluyen apoyo tecnológico de la propia Microsoft en su vertiente Microsoft's Component Object Model (COM), además de nuevos tipos de librerías y la mejora de una unidad central que compila y genera códigos más rápidos y pequeños que antes. También se ha aumentado y perfeccionado el entorno de desarrollo. Hay que destacar asimismo el trabajo a fondo que Microsoft ha llevado a cabo en torno al lenguaje común del ANSI C++. Esta versión 5.0 de la edición profesional de Visual C++ se preveía que estuviera en el mercado el 19 de Marzo de 1997 y formará parte del paquete de Visual Studio 97. Microsoft Visual Studio 97 ofrece a los operadores las herramientas necesarias que les ayudan a obtener un éxito seguro en el nuevo mundo de las aplicaciones de desarrollo. Es además el primer paso para la configuración de un set definitivo de herramientas de desarrollo. Su aparición equivaldrá a disponer de un grupo muy efectivo de herramientas para el desarrollo de las aplicaciones y componentes de C++. Con ellas se accederá a bases de datos lejanas, incluidas las SQL, lo que permitirá a su vez aprovechar las ventajas de las últimas tecnologías de Internet y ActiveX.

LIBROS



Inteligencia Artificial

Coincidiendo con la serie que hemos comenzado sobre Inteligencia Artificial, en este número comentamos un libro atípico en la sección. Inteligencia Artificial no es un libro puramente técnico. Se trata de una obra que proporciona un punto de vista distinto al habitual en el estudio del tema, que presenta sus orígenes y se adentra en la complejidad de los procesos humanos susceptibles de ser tratados bajo la perspectiva de la IA, desde un enfoque sociológico. Nos encontramos ante un libro que puede interesar a aquellos con curiosidad científica, que deseen conocer las causas y los posibles resultados de la investigación que se lleva a cabo en el campo de la IA, y utilizar dichos conocimientos como base para decidir su inclusión entre quienes siguen de cerca los progresos y las decepciones que tienen lugar en la "informática del futuro".

Editorial: Acento Editorial
Autor: Daniel Crevier
376 páginas
Idioma: Castellano

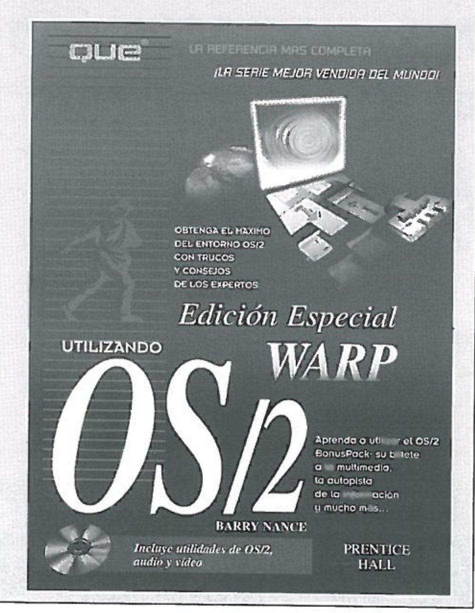
Utilizando OS/2 WARP (Edición Especial)

OS/2 WARP es un sistema operativo de 32 bits para PC capaz de ejecutar software DOS, software Windows y software OS/2. Incluye mejoras internas tanto para aumentar la velocidad como para disminuir los requerimientos de memoria, así como mejoras externas tales como procesador de textos, hoja de cálculo, gestor de base de datos, conectividad con Internet o gestor de información profesional.

Utilizando OS/2 WARP explica con detalle aspectos como la conectividad en red, la naturaleza orientada a objeto de OS/2, la configuración y la solución de problemas a través de temas en los que se tratará la interfaz de Workplace Shell, la interfaz de línea de comando, los objetos de controlador de dispositivo y las miniaplicaciones que trae OS/2.

Como guía de aprendizaje y de consulta puede ser un material de indudable valor donde encontrar un excelente tratamiento de la filosofía que se esconde tras OS/2.

Editorial: Prentice Hall
Autor: Barry Nance
674 páginas (incluye CD-ROM)
Idioma: Castellano



TE BUSCAMOS...

SI TIENES AMPLIOS CONOCIMIENTOS O EXPERIENCIA EN ALGUNO DE ESTOS CAMPOS

- Programación en Visual Basic, Visual C++, Delphi, HTML, JAVA, JAVAScript, CGI
(Ref. PROG)
- Lenguajes multimedia como Director, Authorware, Toolbook
(Ref. MULT)
- Dominio de sistemas operativos DOS, Windows 95, Windows NT, UNIX, Linux, OS2.
(Ref. S.O)
- Redes, Conectividad y Sistemas abiertos
(Ref. RED)
- Diseño y autoedición en CorelDraw, Photoshop, FreeHand, QuarkXPress, Page Maker
(Ref. DIS)
- Infografía con 3D Studio, LightWave
(Ref. INF)
- Internet, Navegación, Videoconferencia, ActiveX, Plug-Ins, Shockwave, RDSI
(Ref. NET)
- Nuevas tecnologías, MMX, USB, DVD, Telefonía móvil
(Ref. TECN)
- Instalación, reparación y mantenimiento de software y hardware
(Ref. INST)
- Hardware: Tarjetas gráficas, Modems, Cámaras digitales, Scaners, Tarjetas de sonido
(Ref. HARD)
- Inteligencia Artificial, Sistemas Expertos, Redes neuronales, Robótica
(Ref. SE)

COLABORA CON NOSOTROS

Envíanos un currículum vitae con la referencia correspondiente a la siguiente dirección:
TOWER COMMUNICATIONS - APARTADO DE CORREOS 54.283 - 28080 MADRID

JAVASCRIPT: CONTROL DE PROGRAMA Y PRIMEROS EFECTOS

Fernando J. Echevarrieta

El mes pasado se realizó una introducción a JavaScript, encuadrándolo en el entorno de las otras tecnologías WWW. Asimismo, se dieron los primeros pasos en este lenguaje con la elaboración de un primer *script* y la presentación del control de eventos JavaScript. En el presente artículo se mostrarán las estructuras de control del lenguaje así como un conjunto de interesantes efectos para añadir a los documentos WWW.

Con esta miniserie de artículos dedicados a JavaScript, primeramente se pretende proporcionar al lector un amplio abanico de herramientas para realizar numerosos efectos en este lenguaje. Así, en una primera etapa, se tratará de cubrir el mayor número posible de los aspectos del mismo buscando como objetivo que desde el primer momento se disponga de una gran cantidad de recursos para realizar documentos llenos de efectos. Esta exposición, basada en ejemplos, será rigurosa, haciendo hincapié en todos aquellos detalles que escapan con facilidad al programador novel de JavaScript y pueden producirle algún quebradero de cabeza. Por otra parte, con la idea de considerar JavaScript como una "extensión" HTML, se procurará hacer lo más sencilla posible la lectura de los artículos, incluso para aquellas personas que carecen de conocimientos de programación.

En una segunda etapa, se profundizará más en el lenguaje, dando consistencia a los recursos que se hayan presentado inicialmente y añadiendo otros más avanzados. Así, a la primera parte que, empleando la terminología de la documentación de Linux, se podría denominar de "HOWTOS", seguirá la segunda que mostrará cómo funciona realmente el lenguaje y, a través de este funcionamiento, se podrá intuir el funcionamiento interno de un cliente de WWW como el Netscape Navigator.

JavaScript, como se justificó en el artículo anterior, es un lenguaje basado en objetos, no orientado a objetos. Pero esto hace que para el lenguaje todo sean objetos. Así pues, JavaScript dispone de un modelo de objetos y de una serie de sentencias de manejo de los mismos, pero no será necesario conocer ni lo uno ni lo otro en esta primera etapa. Por ello, en el presente artículo, se tratarán temas como el URL Javascript, que permite la activación de funciones JavaScript desde enlaces HTML; las diferencias entre las dos versiones de JavaScript, la 1.0 y la 1.1., y entre los browsers que las interpretan, Netscape Navigator y MS Internet Explorer; el empleo de atributos variables en etiquetas HTML, exclusivo de la versión 1.1; la generación de distintos tipos de ventana, de alerta, de confirmación de condiciones y de entrada de datos de usuario; cómo cambiar de documento desde JavaScript y cómo tomar la decisión de qué documento cargar en función de la respuesta a una ventana de confirmación; cómo emplear botones como enlaces; o los factores a considerar a la hora de escribir directamente en un documento desde JavaScript. La mayoría de los ejemplos que figuran en el artículo han sido incluidos en el disco de la revista en el fichero ejemplos.htm, que puede ser consultado con cualquiera de los dos browsers mencionados.

Además, como todo lenguaje de programación, JavaScript dispone de una serie de sentencias para el control de flujo de programa. Aunque realmente no es tampoco necesario conocerlas para comenzar a añadir efectos a nuestros documentos WWW o para entender los ejemplos del artículo, sí lo será en cuanto el lector desee despegarse de la revista y comenzar sus propios programas. Así pues, aunque no

sea la parte más "divertida", se comenzará con una exposición de las sentencias de control de flujo de programa JavaCript.

CONTROL DE FLUJO DE PROGRAMA

JavaScript cuenta con cinco sentencias de control cuya sintaxis resultará muy familiar a todos los programadores en C y Java.

Sentencia if

Es la sentencia de toma de decisiones con la que cuentan todos los lenguajes de programación. Su sintaxis es la siguiente:

```
if (condicion) {
  sentencias
}
else {
  sentencias
}
```

donde la condición será cualquier expresión. En sucesivos ejemplos se irá descubriendo la sintaxis de expresiones válidas en JavaScript.

Sentencia for

La sentencia for es quizá la más potente del lenguaje y se corresponde exactamente con su homónima en C:

```
for (expresión de arranque; condición de mantenimiento; expresión de incremento)
{
  sentencias
}
```

Es importante destacar que entre las expresiones y el carácter de "punto y coma" no debe haber ningún espacio en blanco.

Cuando el flujo de programa JavaScript llega a una sentencia for, sigue los siguientes pasos:

1. Ejecución de la expresión de arranque.
2. Ejecución de las sentencias incluidas en el bucle (rodeadas por llaves) siempre que se cumpla la condición de mantenimiento.
3. Tras cada iteración, ejecución de la expresión de incremento.

Por ejemplo, para realizar un programa que muestre los cuatro primeros números naturales elevados al cuadrado (figura 1), se podría teclear en el cuerpo de un documento HTML:

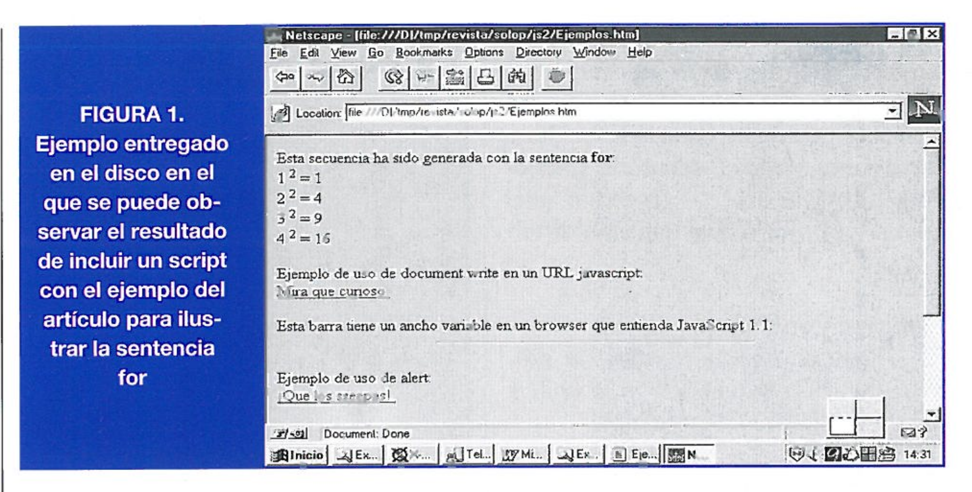


FIGURA 1. Ejemplo entregado en el disco en el que se puede observar el resultado de incluir un script con el ejemplo del artículo para ilustrar la sentencia for

```
<SCRIPT>
<!-- Ocultando
for (i=1; i<5; i++)
{
  document.write(i + " <SUP>2</SUP> = " + i*i + "<BR>")
}

// -->
</SCRIPT>
```

Así, al comenzar, JavaScript asigna a la variable i el valor 1 y mientras el valor de i es menor que 5, ejecuta las sentencias rodeadas por llaves y tras cada ejecución, ejecuta la expresión de incremento, en este caso i++. Como se puede observar, al contrario que en lenguajes como C, C++ o Java, no es necesario declarar una variable ni su tipo antes de usarla. Así, al encontrar la asignación i=1, el intérprete generará dinámicamente la variable i de tipo numérico.

El formato de las expresiones sí es, en cambio, idéntico al empleado en los mencionados lenguajes. Así, i++ es un post-incremento y equivale a la asignación: i=i+1.

Otro operador que se observa en el ejemplo es el de concatenación de cadenas de texto, que se representa mediante el carácter +. Así, "Sólo" + "Programadores" equivale a "Sólo Programadores" y lo mismo ocurre si se emplea una variable, por ejemplo: "Sólo " + cadena, habiendo indicado previamente cadena= "Programadores".

Sentencia while

Esta sentencia es la más simple para realizar bucles sujetos a una condición. Su sintaxis se corresponde con:

```
while (condicion)
{
  sentencias
}
```

Si la condición inicial no es cierta nunca, no se producirá ninguna iteración. Si, por el contrario, es cierta siempre, se conseguirá un bucle infinito. Por ejemplo:

```
while (true) {
...
}
```

En este ejemplo se ha empleado la constante true. JavaScript admite el tipo booleano de datos que consta de dos únicos valores: true y false

Sentencia break

Esta sentencia se emplea para interrumpir de manera "anormal" un bucle while o for. Por ejemplo:

```
i=0
while (i < 6) {
  if (i == 3)
    break
  i++
}
```

define un bucle que asigna los valores 0, 1, 2 y 3 a la variable i, aunque la condición del mantenimiento del bucle while indique que se debe llegar hasta 6. Cuando la variable alcanza el valor 3, se ejecuta la sentencia break que da por finalizado el bucle.

En el ejemplo se puede observar también una expresión de comparación, que como en C o Java se realiza mediante una doble igualdad ==. Otras expresiones válidas serían < (menor que), > (mayor que), <= (menor o igual que), >= (mayor o gual que), != (distinto de).

Sentencia continue

La sentencia continue termina la presente iteración de un bucle for o while y pasa a la siguiente. Por ejemplo:

```
siempre=0
aveces=0
for (i=1 ; i<=6, i++) {
  siempre++
  if (i==2) continue
  aveces++
}
```

Este ejemplo incrementa el valor de la variable siempre en cada iteración. Sin embargo, cuando i alcanza el valor 2, la sentencia continue provoca que se salte directamente a la siguiente iteración, sin terminar la actual. Por ello, en ese caso, la variable aveces no se incrementa. Así, al finalizar el bucle, la variable siempre valdrá 6 y la variable aveces, 5. Así pues, la diferencia con la sentencia break es que aquella finalizaba el bucle completamente, mientras que continue finaliza únicamente la iteración actual, prosiguiendo el bucle con la siguiente.

Con estas sentencias, es posible emplear JavaScript como un lenguaje de programación estructurada. JavaScript también cuenta con sentencias de manejo de objetos, pero se dejarán para más adelante. De momento, el lector no acostumbrado a la programación con objetos se irá familiarizando con su terminología a través de los ejemplos.

EL URL JAVASCRIPT (COMO EXCUSA PARA CONTAR OTRAS COSAS)

Una de las aportaciones más interesantes que ofrece JavaScript es la inclusión de un nuevo URL: el URL javascript. Al igual que URLs como mailto, se trata de un URL falso, pero es una de las ideas que más potencia le dan al

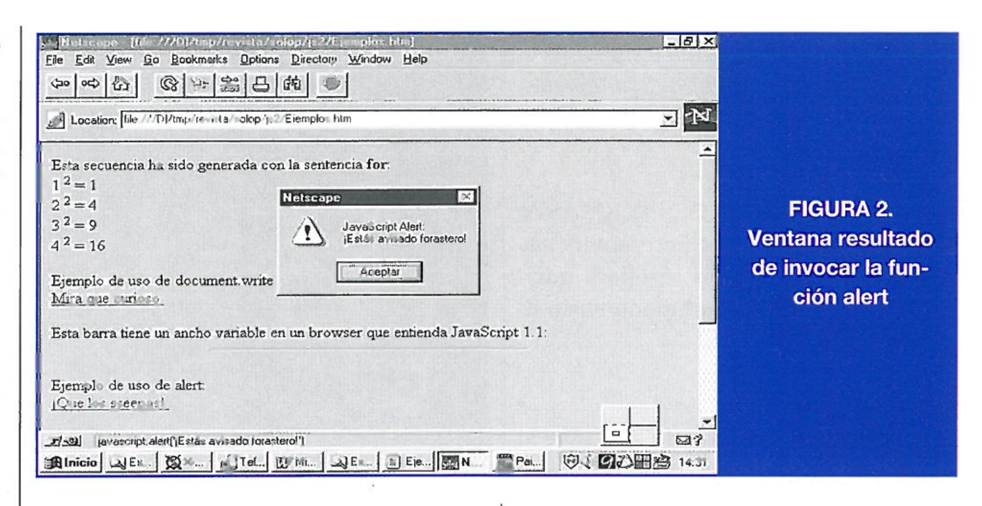


FIGURA 2. Ventana resultado de invocar la función alert

lenguaje, ya que permite incluir cualquier función JavaScript en un enlace HTML. Para ello, se indica del siguiente modo:

```
<A HREF="javascript:funcion()">
Texto del enlace
</A>
```

donde funcion() es la función JavaScript en cuestión.
Por ejemplo :

```
<A HREF="javascript:escribe
('HOLA<BR>')">
Mira que curioso
</A>
```

donde escribe se ha definido en la cabecera de la forma:

```
function escribe(texto)
{
  document.write(texto)
}
```

Los ejemplos a veces dicen más de lo que parece y éste, por ejemplo, dice lo siguiente:

- Como se puede observar, se ha incluido el URL JavaScript entre comillas dobles ("). Esto no suele ser necesario en HTML, pero conviene siempre rodear los valores de los atributos de las etiquetas por este tipo de comillas para asegurarse de que todos los parsers (analizadores) entenderán correctamente el código.
- Al rodear un atributo con comillas dobles, el parser las considerará un carácter delimitador, por lo que ya no se podrán emplear como tal en la expresión que se pasa a la función JavaScript. Por ello, se han empleado comillas simples, que también son admitidas.
- La función document.write borra la ventana del navegador y escribe la expresión que se le pasa en una nueva página en blanco. Esto se debe a que, una vez se ha cargado y presentado un documento HTML, ya no se puede hacer nada para cambiarlo. Por ello, si se añade nuevo código al mismo se presenta en una pantalla en blanco. El primer programa en JavaScript que se realizó en el artículo del pasado número también empleaba document.write, y el texto aparecía mezclado con el código HTML del documento. Pero esto ocurría porque el script se encontraba incrustado en el código mediante la etiqueta <SCRIPT>, por lo que se ejecutaba antes de que el documento terminara de cargarse y mostrarse. Así, el navegador leía código HTML y lo mostraba, llegaba a la etiqueta <SCRIPT>, interpretaba el código y mostraba el resultado, y seguía leyendo código HTML hasta que terminaba de cargar el documento.
- La expresión que se ha pasado a document.write termina con el símbolo HTML de fin de línea:
. Esto se debe a que, al escribirse la expresión sobre una ventana en blanco, y no formando parte de un texto HTML, el navegador no mostrará una línea hasta que no esté completa, y si no se envía código HTML que indique fin de línea, como <HR> o <P>, el browser seguirá esperando más información para



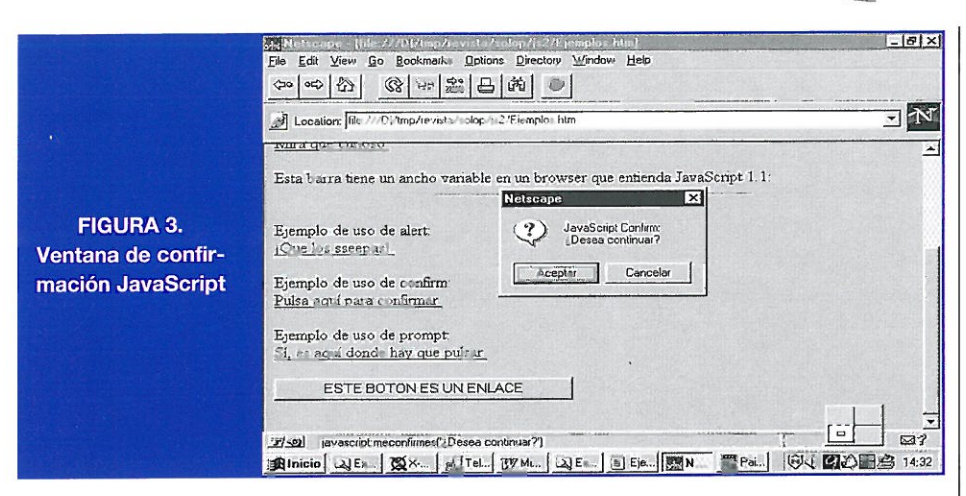


FIGURA 3. Ventana de confirmación JavaScript

completar la línea. No sirve de nada enviar un carácter ASCII de retorno de carro "\n", ya que, como recordará el lector, HTML ignora los retornos de carro ASCII.

- Algún lector puede estar pensando en el motivo por el que se ha definido una función tan "tonta" como escribe, que lo único que hace es llamar a document.write y no se ha empleado document.write directamente, por ejemplo:

```
<A HREF="javascript:escribe
('HOLA<BR>')">
```

Si bien es verdad que el autor de este artículo es retorcido, nada se hace sin razón, y es que, en JavaScript, como en muchos otros lenguajes, durante la ejecución, una invocación a una función es directamente sustituida por su valor de retorno. El problema surge del hecho de que document.write devuelve un valor de retorno, a saber, true o false si tiene éxito o fracasa respectivamente. Así pues, si se indicara este método directamente en el URL javascript, se obtendría un estupendo "true" en pantalla, en lugar del deseado texto "HOLA". Realmente, la función sí habría escrito la cadena deseada, pero inmediatamente habría sido tapada por el texto "true". Puede ser interesante hacer la prueba, y otra prueba interesante es modificar la función escribe para que devuelva un valor de retorno. Por ejemplo:

```
function escribe(texto)
{
  document.write(texto)
  return 5
}
```

Estos son los pequeños detalles que no se cuentan en ningún sitio y pueden desesperar al principiante JavaScript al ver que algo "tan sencillo" no funciona.

ESTO PARECE HTML (O LAS ENTIDADES JAVASCRIPT)

Es un buen momento para advertir al lector que "circulan por ahí" dos versiones diferentes de JavaScript, las versiones 1.0 y 1.1. JavaScript es una propuesta de Netscape, que lo incluyó en la versión 2.0 de su browser Netscape Navigator, por aquel entonces líder indiscutible de los navegadores de Internet, seguido muy por detrás por MS Internet Explorer. Pero más tarde, la batalla entre ambas compañías se desató, y MS Internet Explorer, en su versión 3.0, dio un salto gigantesco hacia adelante para poder competir en igualdad de condiciones con Navigator.

Las llamadas a la función confirm de JavaScript se están haciendo cada vez más "populares" en Internet

Así, le fueron incorporadas todas las facilidades de que este último disponía junto a otras nuevas. Entre ellas se incluyó el intérprete de JavaScript. Pero Navigator tuvo tiempo de evolucionar, por lo que su versión 3.0 incorporó una nueva versión de JavaScript, la versión 1.1. Explorer; en esto va un paso por detrás y, aunque el autor no ha tenido tiempo de probar todas las nuevas incorporaciones de JavaScript 1.1 en Explorer 3.0, todas las que ha probado le han fallado. Así pues, parece que este browser, sólo implementa la versión 1.0. En cualquier caso, el autor aún no ha tenido acceso a la versión de 3.1 de Explorer, pero si ésta no entiende JavaScript 1.1, es seguro que una próxima versión lo hará, ya que JavaScript 1.1. incorpora capacidades tan notables como la posibilidad de cambiar las imágenes de un documento ya cargado en respuesta a un evento. Sea como fuere, en esta serie se comentará la versión "oficial" de Netscape de JavaScript, y siempre que se trate con alguna característica especial de la versión 1.1, se hará constar.

Este es el caso, por ejemplo, de las entidades JavaScript. Como recordará el lector, en HTML se definían una serie de "cosas" (por evitar la redundancia) que se denominaban entidades HTML. Estas entidades, que representaban caracteres que presentaban conflicto con los empleados por HTML, como "<" ó ">", o caracteres especiales, como acentos, se caracterizaban por comenzar por el símbolo ampersand (&) y terminar por un punto y coma (;), encerrando ambos el nombre de la entidad, por ejemplo é para una é (e con acento agudo). Las entidades JavaScript, responden a la misma sintaxis pero, para diferenciarlas de las entidades HTML, además, encierran el nombre entre llaves ({}). Su razón de existir es que permiten introducir valores variables controlados por JavaScript como atributos HTML. Así, por ejemplo, si se define una variable ancho en JavaScript, podrá ser posible definir barras de ancho variable en HTML de la forma:

```
<HR SIZE="&{ancho};">
```

VENTANAS DE DIÁLOGO Y CARGA DE URLs

Uno de los efectos que más llaman la atención, por su interactividad, en un documento WWW, es la aparición de ventanas de diálogo como respuesta a un evento. JavaScript proporciona un completo mecanismo para el control de

ventanas y frames del navegador, pero no es necesario conocerlo para poder generar las ventanas de uso más habitual, como ventanas de alerta, de confirmación o de entrada de datos. Para ello, proporciona las siguientes funciones predefinidas:

alert(mensaje)

Muestra una ventana de alerta, generada por el manejador de ventanas de nuestro sistema operativo (es decir, no es una ventana de navegador), con el mensaje que se le pasa como parámetro y un botón de confirmación que al ser pulsado cierra la ventana. Un ejemplo tan simple como poco útil podría ser:

```
<A HREF="javascript:alert('Estás avisado forastero!')">
Que lo sseepas!
</A>
```

cuyo resultado puede observarse en la figura 2.

confirm(mensaje)

Una función mucho más útil, que actúa como alert pero proporcionando al usuario dos botones, uno para confirmar y otro para rechazar. La ventaja de esto es que la función devuelve un valor de retorno, true o false respectivamente, que puede ser empleado en un script. Por ejemplo, el enlace siguiente llama a la función meconfirmes, que hace uso de la función JavaScript confirm:

```
<A HREF="javascript:meconfirmes
('¿Desea continuar?')">
Pulsa aquí para confirmar
</A>
```

La función meconfirmes habrá sido previamente definida en la cabecera del documento, como se recomendaba en el artículo del mes pasado:

```
function meconfirmes(ello)
{
  if (confirm(ello))
        document.location="http://highland.dit.upm.es:8000"
  else
document.location ="http://www.disney.com"
}
```

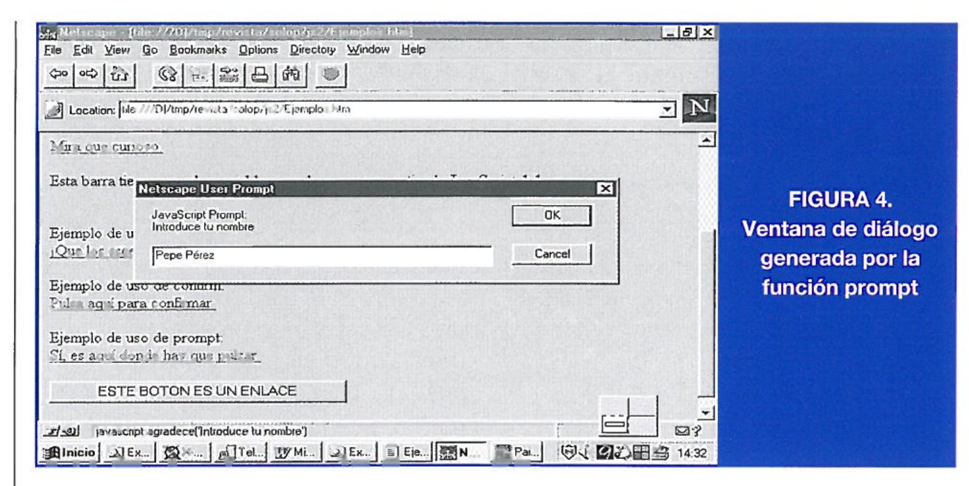


FIGURA 4. Ventana de diálogo generada por la función prompt

Este es el típico recurso que se emplea en los servidores de Internet "para adultos", en los que al cargar la página principal se advierte que contienen imágenes que pueden herir la sensibilidad y se pide confirmación mediante esta función para seguir avanzando. En estos casos, la ventana de aviso suele aparecer automáticamente al entrar en un servidor. Para ello, se hace uso de la segunda relación entre JavaScript y HTML que se comentaba en el artículo del mes anterior: la gestión de eventos mediante inclusión de manejadores en etiquetas HTML. En este caso, el browser, al cargar una página, genera un evento denominado Load, cuyo manejador puede ser insertado en la etiqueta <BODY> de la forma:

```
<BODY onLoad="javascript:meconfirmes('¿Desea continuar?`) ;return true">
```

Ahora, en función de la respuesta, será posible enviar al usuario a un lugar u otro de la Internet. Para ello, como se puede observar en el código de la función meconfirmes, se asigna a document.location el valor del URL a cargar. Ya el mes pasado, se tuvo un primer contacto con el objeto document, cuando se escribía en pantalla mediante document.write. Entonces se vio cómo write era un método del objeto document, aunque document.write podía ser considerado una función más por aquellos programadores JavaScript que no quisieran saber nada de OOP (programación orientada a objetos). En esta ocasión, nos encontramos con una propiedad del objeto en lugar de un método: location. Las propiedades de los objetos se manejan como variables, para su consulta basta hacerlas formar parte de una expresión, por ejemplo:

```
document.write(document.location)
```

y para modificarlas, basta asignarles un valor, como se ha hecho en la función meconfirmes. Pero al asignar un valor a una propiedad de un objeto se cambia la propiedad, lo que afecta al objeto. Así, en este caso, el objeto document, que engloba todo lo relativo al contenido de una ventana del navegador, cam-

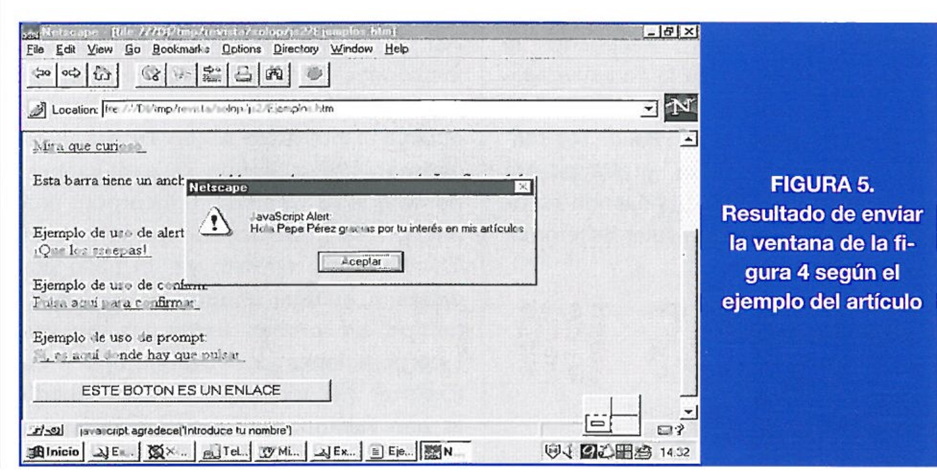


FIGURA 5. Resultado de enviar la ventana de la figura 4 según el ejemplo del artículo



bia su URL, es decir, se carga una nueva URL automáticamente ¡y para ello sólo ha hecho falta una asignación!

Se recuerda también al lector que cuando se llama a una función JavaScript desde un navegador, es necesario incluir un valor de retorno, por lo que se ha incluido " ;return true" en el ejemplo. Aquí tambien se puede observar otra característica del lenguaje que puede haber llamado la atención a algún lector: el empleo del "punto y coma" (;). Al contrario que en lenguajes como C, en los que es obligatorio terminar cada sentencia del lenguaje con este símbolo, que actúa como delimitador, en los ejemplos anteriores se puede observar que no se ha empleado nunca. Y es que JavaScript considera como delimitador el propio avance de línea. Sin embargo, en el último ejemplo, con el manejador onLoad, se han incluido dos sentencias JavaScript en una misma línea. En este caso sí es necesario emplear este delimitador. En cualquier caso, aquellos que estén acostumbrados a incluirlo siempre podrán hacerlo sin problemas.

```
prompt(mensaje,valor_por_defecto)
```

El tercer tipo de ventana especial ofrecida por JavaScript es una ventana de diálogo en la que se ofrece al usuario un campo a través del cual puede introducir datos al programa JavaScript y dos botones, uno de aceptar y otro de cancelar. Esta ventana muestra el texto indicado por mensaje y, por defecto, ofrece el campo ya relleno con el valor_por_defecto.

Por ejemplo, supóngase el enlace:

```
<A HREF="javascript:agradece
('Introduce tu nombre')">
Sí, es aquí donde hay que pulsar
</A>
```

para el que la función agradece se ha definido en la cabecera como:

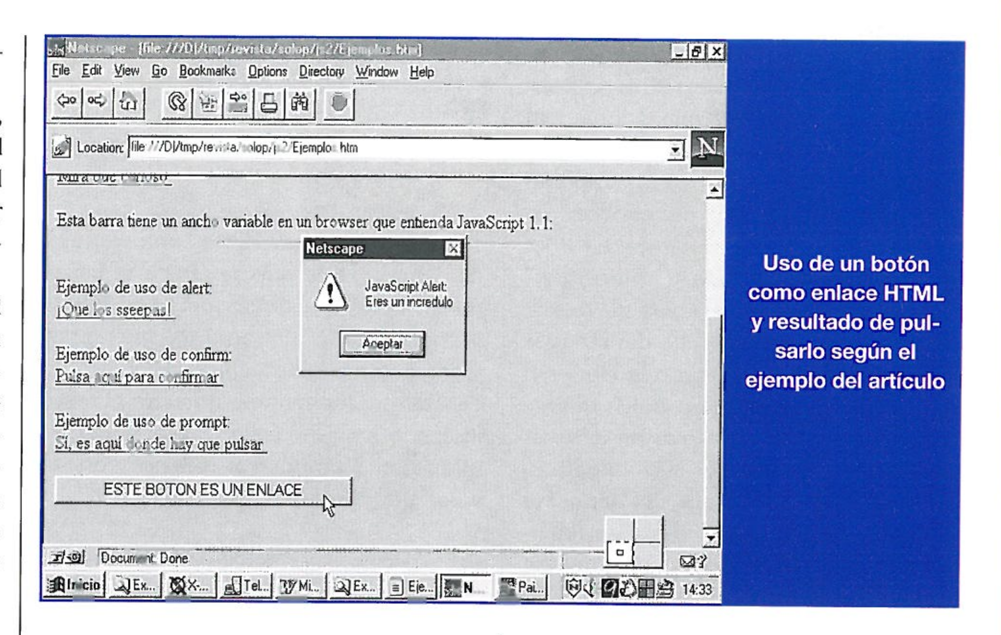
```
function agradece(texto)
{
  alert("Hola " + prompt(texto,'') + " gracias por tu interés en mis artículos")
}
```

La pulsación del enlace producirá un resultado como el que se observa en las figuras 4 y 5. Sobre este código se pueden hacer algunos comentarios:

- En primer lugar, se puede observar cómo no se llama directamente a la función prompt, sino que se ha introducido ésta como parte de la expresión que se pasa como parámetro a alert. Por supuesto, se podría haber almacenado el valor de retorno en una variable y haberla pasado a alert, pero, como se ha comentado anteriormente, durante la ejecución, una invocación a una función es directamente sustituida por su valor de retorno. Así, aunque primeramente aparezca la función alert, el intérprete de JavaScript deberá ejecutar primero la función prompt para disponer de una sentencia completa. Así pues, primero apare-

cerá la ventana de prompt y después la de alert.

- La función, como era de esperar, retorna el valor introducido por el usuario o null si éste pulsa sobre el botón de cancelar. Así que, por favor, en el ejemplo, no pulse cancelar:-)
- Otra anotación a realizar es el hecho de que se ha pasado una cadena vacía como valor por defecto a la función prompt. En caso de no indicar valor por defecto aparece un "horroroso" texto "<undefined>", por lo que, si se desea ofrecer un campo vacío, será necesario indicar una cadena de texto vacía.



Uso de un botón como enlace HTML y resultado de pulsarlo según el ejemplo del artículo

EMPLEANDO BOTONES COMO ENLACES

Otra de las facilidades que proporciona el mecanismo de manejo de eventos de JavaScript es la posibilidad de emplear botones de un form como enlaces HTML, proporcionando a las páginas un look similar al que presentaría un applet Java.

Originalmente, en HTML 2.0 existían únicamente dos tipos de botones, que se lograban mediante la asignación de los valores SUBMIT o RESET al atributo TYPE de la etiqueta <INPUT> (todas las etiquetas HTML 2.0 correspondientes a la definición de forms pueden consultarse en el artículo "El lenguaje HTML (II)" de esta misma serie). Por ejemplo:

```
<INPUT TYPE="SUBMIT">
```

El primer tipo provocaba el envío de la información al servidor mientras que el segundo borraba todos los campos del form. Posteriormente, se amplió el lenguaje para incluir un nuevo tipo INPUT denominado BUTTON, cuyo uso es el siguiente:

```
<INPUT TYPE="BUTTON" VALUE=
texto_boton [...]>
```

donde texto_boton es el texto que aparecerá sobre el botón y [...] representa el resto de los atributos que se pueden incluir en la etiqueta <INPUT>.

Se recuerda al lector que las etiquetas correspondientes a elementos de un form, sólo tenían sentido cuando se encontraban rodeadas por la etiqueta <FORM>...</FORM>. Navigator incluso las ignorará si no se incluye esta última etiqueta (esto es consecuencia directa del modelo de objetos del navegador, que se estudiará en un futuro artículo).

Así pues, la primera idea que se le ocurre a uno para emplear un botón como enlace es rodear al mismo con la etiqueta </A>:

```
<FORM>
<A HREF="http://
highland.dit.upm.es:8000>
<INPUT TYPE="BUTTON"
VALUE="ESTE BOTON ES UN ENLACE">
</A>
</FORM>
```

Esto, sin embargo, producirá un botón y a su lado un enlace sin ancla, que apenas será visto y es que, desde HTML, no se puede lograr este efecto. Para lograrlo será necesario, una vez más, hacer uso de la gestión de eventos de JavaScript, simplemente añadiendo el manejador onClick (que detecta la pulsación de ratón sobre un objeto) al botón en cuestión:

```
<FORM>
<INPUT TYPE="BUTTON"
VALUE="ESTE BOTON ES UN ENLACE"
onClick="alert('Eres un
incrédulo');return true">
</FORM>
```

El resultado producido por este código se puede observar en la figura 6.

CONCLUSIONES

Con el presente artículo se ha facilitado ya un interesante conjunto de herramientas JavaScript para incluir en los documentos WWW del lector. En próximos artículos se presentarán nuevos mecanismos y se realizarán programas más complejos. Asimismo, se presentará el modelo de objetos del lenguaje y el funcionamiento de un navegador WWW, mostrando la fuerte relación que hay entre ambos.

CONTACTAR CON EL AUTOR

Para cualquier duda, comentario, sugerencia o crítica relacionadas con el artículo, se anima al lector a que se ponga en contacto con el autor a través de:

E-mail Internet: echeva@dit.upm.es
E-mail CompuServe: 100646,2456
WWW: http://highland.dit.upm.es:8000.

INTERRUPCIONES

José C. Remiro

El presente artículo pretende mostrar el funcionamiento del mecanismo para el tratamiento de interrupciones hardware y el conjunto de elementos de programación e instrucciones que pueden utilizarse para realizar una rutina para el tratamiento de éstas. Además, y puesto que para realizar tales rutinas es necesario conocer las características del hardware, se presentará un conjunto de procedimientos y funciones que permiten controlar el puerto serie, tema que se trató en el anterior artículo de esta sección.

Una interrupción hardware no es más que una señal que le llega a la CPU desde un dispositivo físico indicando que éste necesita su atención. Para no hacer depender en demasía la CPU del hardware, las interrupciones de este tipo son tratadas en primer lugar por el controlador de interrupciones, un circuito que se encarga de recibirlas y presentarlas a la CPU según el orden de llegada, o bien en el caso de haber llegado de forma simultánea, entregarlas según una serie de prioridades. Éstas están fijadas de antemano por el diseño de la placa. En los PCs actuales están disponibles 16 interrupciones hardware que tienen asociadas como prioridad un número entre 0 y 15, representando el 0 la de mayor prioridad.

Una vez que el controlador de interrupciones ha decidido comunicar el número de interrupción que se ha producido, envía a la CPU una señal; ésta almacena el contenido de los siguientes registros del proceso que estaba ejecutando, a saber, el contador de programa, los registros de indicadores y el registro de segmento de código. De esta forma la CPU puede pasar a ejecutar la rutina asociada a la interrupción (ésta es independiente del proceso que se estaba ejecutando); una vez que se termina de ejecutar, la CPU recupera los valores almacenados anteriormente y prosigue la ejecución del proceso que quedó pendiente.

También es posible disponer de interrupciones software, su funcionamiento es similar, aunque el origen de la interrupción no es una señal sino una instrucción máquina (INT). Como ya es sabido, este tipo de interrupciones puede utilizarse como un mecanismo para la interacción entre un programa y diferentes rutinas de la ROM-BIOS.

La forma en que la CPU localiza la rutina que da servicio a la interrupción es la siguiente. Todas las interrupciones disponen de un número que las identifica, en total hay 256 interrupciones, numeradas de 0 a 255. Si la interrupción es hardware, el controlador de interrupciones es el que comunica a la CPU el número de la interrupción; si la interrupción es software, la propia instrucción tiene como parámetro el número de interrupción. Los primeros 1024 bytes del mapa de memoria de un PC están reservados para los vectores de interrupción, un vector de interrupción no es más que un puntero que señala la primera instrucción donde comienza la rutina de tratamiento de interrupción que le corresponde. Puesto que una dirección está compuesta por un número de segmento y un número de desplazamiento, cada vector de interrupción ocupará 4 bytes. Así, si la CPU sabe el número de interrupción a tratar, basta con que multiplique por 4 el número de interrupción para conocer la dirección de servicio de dicha interrupción; si carga los valores contenidos en dicha dirección dentro de los registros

Es posible la construcción de rutinas de tratamiento de interrupción mediante lenguajes de alto nivel, sin necesidad de recurrir al lenguaje ensamblador.

CS e IP entonces iniciará la ejecución de la rutina.

El mecanismo para el tratamiento de interrupciones puede parecer a primera vista bastante complicado, pues bastaría con que a cada interrupción se le asignase de forma directa la dirección de comienzo de la rutina que tiene asociada; sin embargo, la disposición original hace muy sencilla la modificación de la rutina de tratamiento de interrupción, pues si al inicializarse el ordenador se dispone en el mismo lugar la dirección del vector de interrupciones, modificando su contenido (la dirección a la que debe saltar la CPU para tratar la interrupción) se habrá modificado la forma de tratarla.

LA TABLA DE VECTORES

Aunque es posible inspeccionar la tabla de vectores utilizando el programa *Debug*, se ha construido un programa en TurboPascal (que puede encontrarse en el CD-ROM que acompaña a la revista) llamado *muestra_vectores*, que permite la inspección de dichos vectores (en los cuadros 1 y 2 pueden observarse los procedimientos más importantes de dicho programa, *visualiza_memoria* y *visualiza_vector*). Los anteriores procedimientos hacen uso del procedimiento *Getintvec* perteneciente a la unidad Dos de Turbo Pascal. Este procedimiento admite como primer parámetro un valor de tipo byte que contiene el número de interrupción cuyo vector se desea obtener. El segundo parámetro no es más que un puntero genérico que albergará tras la llamada al procedimiento la dirección contenida en el vector.

Aunque es posible el acceso a memoria a través de los arrays predefinidos *Mem, Memw y Meml*, que utilizan como índice, separados por dos puntos, la base y el desplazamiento del segmento de la posición de memoria que se desea acceder, este acceso es indeseable para la zona de memoria que alberga los vectores. Supóngase que se desea modificar la dirección a la que apunta un vector de una determinada interrupción y que ya está modificada la base y que en el momento en que se va a actualizar el desplazamiento se produce una llamada a la interrupción que se está modificando. La CPU saltará a una dirección de memoria que realmente no albergará la rutina de tratamiento para dicha interrupción, por lo que con toda probabilidad caerá el sistema. Afortunadamente, las funciones 25h y 35h del DOS permiten manipular los vectores de interrupción de forma correcta, pues antes de proceder a su lectura o modificación se encargan de desactivar las interrupciones, volviendo a activarlas cuando han terminado su función. Realmente el procedimiento *Getintvec* no es más que una llamada a la función 35h del DOS, siendo *Setintvec* el procedimiento correspondiente a la función 25h, del que se hablará posteriormente.

El programa *muestra_vectores* se compone básicamente de un bucle que permite al usuario en cada iteración introducir un número de interrupción o bien finalizar la ejecución del programa. Mediante el procedimiento *visualiza_vector*, se accede al contenido del vector de interrupciones obteniendo la dirección de la rutina de tratamiento para dicha interrupción. Hay que tener en cuenta a la hora de formatear los datos de salida, que la memoria del PC está direccionada en unidades de 8 bytes y que cuando se trabaja con palabras de 16 bytes (como en el presente caso) los dos bytes se almacenan adyacentes pero en orden inverso; por esta razón, el *array* que almacena las direcciones (*dir_aux*) es visualizado en orden inverso.

El procedimiento *visualiza_memoria* realiza un volcado hexadecimal de los primeros 256 bytes adyacentes a la dirección de memoria indicada en el vector para la interrupción seleccionada. Si el primer byte contuviese el valor CFh, se correspondería con un retorno de interrupción. Que una interrupción apunte a dicha instrucción es una medida de seguridad para que no pueda producirse la activación de dicha interrupción si el vector no ha sido inicializado previamente. El efecto que produce un retorno de interrupción es la devolución del control de ejecución al punto donde se realizó la llamada. Además, este procedimiento intenta visualizar aquellos bytes que se corresponden con caracteres ASCII, por lo que en ocasiones podrá observarse parte de los créditos y de las firmas de propiedad de la rutina de servicio de la interrupción.

LOS PUERTOS DE E/S

La CPU se comunica con la mayoría de los dispositivos físicos, como el teclado o la impresora, a través de los puertos de E/S. Cada puerto está identificado por un número de puerto de 16 bits, dentro del rango 00h a FFFFh. Cada puerto está asignado a un único dispositivo, pero cada dispositivo puede tener asignados varios puertos, dependiendo de su propósito. Al igual que sucede cuando la CPU desea acceder a la memoria, se utilizan los buses de datos y direcciones para la comunicación con los puertos. En primer lugar la CPU manda una señal a través del bus del sistema para indicar que la dirección que va a mandar por el bus se corresponde con la de un puerto. La CPU envía la dirección del puerto, siendo el dispositivo al que esté asociada

Cuadro 1.

```
procedure visualiza_memoria (num_int: integer);
 var
  dir_rutina: pointer;
  ptro_mem: ^memoria;
  mem_aux: memoria;
  i, j: integer;
 begin
  getintvec (num_int, dir_rutina);
  ptro_mem:= dir_rutina;
  mem_aux:= ptro_mem^;
  writeln;
  if mem_aux[0] = $CF
    then
     writeln ('El vector apunta a un retorno de
interrupci¢n')
    else
     begin
      for i:=0 to 15
        do
         begin
         for j:=0 to 15
           do
            begin
             byte_a_hexa(ord(mem_aux[i*16+j]));
             write(' ');
            end;
          write (' [');
          for j:=0 to 15
            do
             begin
              ch:= chr(mem_aux[i*16+j]);
              if (ord(ch) > 31) and (ord(ch) < 127)
               then
                write(ch)
               else
                write('_');
             end;
          writeln(']');
         end;
     end;
 end;
```

Cuadro 2.

```
var
 dir_rutina: pointer;
 auxiliar: ^byte;
 dir_aux: direccion;
begin
 getintvec(num_int, dir_rutina);
 auxiliar:= dir_rutina;
 dir_aux:= direccion(auxiliar);
 write (num_int:3, ' = $');
 byte_a_hexa(num_int);
 write(' DIRECCION -> [');
 byte_a_hexa(dir_aux[3]);
 byte_a_hexa(dir_aux[2]);
 write(':');
 byte_a_hexa(dir_aux[1]);
 byte_a_hexa(dir_aux[0]);
 write(']')
end;
```

dicha dirección el que responda a la llamada de la CPU.

Es importante distinguir entre una dirección de memoria y una dirección de puerto. Las direcciones de puerto no tienen asignada una dirección de memoria, solamente está asociada con un dispositivo de E/S. Así, el puerto con dirección 2A8h no es equivalente a la dirección de memoria 002A8h. Esta diferencia también se pone de manifiesto en las instrucciones y estructuras para acceder a los puertos. En Turbo Pascal las estructuras asociadas al acceso a los puertos de E/S son *Port* y *Portw* (recuérdese que las estructuras para el acceso directo a memoria son *mem, meml* y *memw*); ambas estructuras son *arrays* unidimensionales, representando cada elemento un puerto de datos, cuya dirección de puerto se corresponde con el índice. El tipo de índice es entero *word*, mientras que cada uno de los componentes del *array* son de tipo byte (para *Port*) y *word* (para *Portw*).

La utilización de estos *arrays* es muy sencilla: cuando se asigna un valor a una de sus componentes se está dando salida a dicho valor por el puerto seleccionado. Cuando se referencia a una componente de dicho *array* en una expresión, se da entrada a dicho valor a través del puerto seleccionado. Hay que tener en cuenta que aunque dichas estructuras son *arrays* no se permite la referencia conjunta de todas sus componentes, ni tampoco se puede asignar a dicho *array* ni a ninguna de sus componentes un parámetro por referencia.

LA DIRECTIVA INTERRUPT

Como ya se comentó con anterioridad, cuando se produce una interrupción se notifica a la CPU. Ésta guarda el contenido de sus principales registros en la pila y calcula, conocido el número de interrupción que se ha producido, la entrada correspondiente en el vector de interrupciones y obtiene de ahí la dirección donde comienza la rutina de tratamiento de la interrupción. Parece ser que si se desea construir una rutina de tratamiento para una interrupción mediante un lenguaje de alto nivel como Turbo Pascal, el programador deberá utilizar el lenguaje ensamblador para realizar estas tareas. Afortunada mente, la librería en tiempo de ejecución y el código generado por el compilador de Turbo Pascal (al igual que otros compiladores) son totalmente interrumpibles. Además, la mayoría de la librería en tiempo de ejecución es reentrante, lo que permite escribir rutinas de servicio de interrupción.

El código de una rutina se dice que es reentrante cuando puede ejecutarse desde dentro de ella misma, normalmente porque la secuencia de código original es interrumpida por algún tipo de sistema. Que una rutina no sea reentrante normalmente está causado porque el código trabaja con tablas estáticas localizadas en la memoria; cuando una rutina no reentrante se está ejecutando y es llamada de nuevo, toda la información guardada en dichas tablas es modificada, impidiendo que cuando regrese al punto en que quedó pendiente su primera ejecución pierda sus valores originales. Hay que indicar que las rutinas del DOS son no reentrantes por lo que cualquier rutina de tratamiento de interrupción no debería utilizar ninguno de los servicios proporcionados por el DOS.

Volviendo a los procedimientos a los que acompaña la declarativa *interrupt*, éstos se declaran siguiendo la estructura que muestra el cuadro 3. Los registros se pasan como pseudoparámetros y pueden ser utilizados dentro del procedimiento. Se pueden omitir uno o todos los procedimientos, comenzando por *Flags* y avanzando hacia *BP*. Es erróneo declarar más parámetros que los indicados y no se puede omitir un parámetro sin omitir los que le preceden.

También en el cuadro 3 se puede observar el conjunto de instrucciones que se ejecutan cuando comienza la ejecución de un procedimiento *interrupt*. En la secuencia etiquetada como prólogo se salvan automáticamente todos los registros en la pila, independientemente de

Cuadro 3.

Declaración de un procedimiento interrupt

```
procedure nom_proc (Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES, BP: word);
  interrupt;
   begin
    .
    .
  end;
```

Prólogo

```
PUSH AX
PUSH BX
PUSH CX
PUSH DX
PUSH SI
PUSH DI
PUSH DS
PUSH ES
PUSH BP
MOV BP, SP
SUB SP, tamaño
MOV AX, SEG DATA
MOV DS, AX
```

Epílogo

```
MOV BP, SP
POP BP
POP ES
POP DS
POP DI
POP SI
POP DX
POP CX
POP BX
POP AX
IRET
```

que estén o no declarados en la cabecera del procedimiento. Se puede observar además, que no existe una instrucción STI para habilitar interrupciones adicionales; como se verá más adelante es posible utilizar la instrucción *inline($FB)* para modificar este comportamiento. En la secuencia etiquetada como epílogo (ver cuadro 3) puede observarse la secuencia de instrucciones que se ejecuta cuando finaliza la ejecución de un procedimiento *interrupt*. Ésta se corresponde con el establecimiento de los registros que fueron guardados en el prólogo del procedimiento, finalizando con una instrucción de retorno de interrupción (IRET).

Aunque no se utilizará en el programa que acompaña al artículo la posibilidad de modificar los parámetros de un procedimiento *interrupt* es posible hacerlo; además es interesante utilizar esta posibilidad cuando se desee implementar una rutina de gestión de interrupciones que tenga similares características a las de los servicios INT 21h del DOS.

UN EJEMPLO

El archivo *ejemplo.pas*, incluido en el CD-ROM, contiene una serie de procedimientos y funciones que pueden utilizarse para realizar un programa con el que comunicar de forma muy básica dos ordenadores a través del segundo puerto serie. Básicamente se compone de una cola circular en la que se almacenan todos los caracteres que van llegando al puerto serie, permitiendo, posteriormente y cuando el programa pueda, presentar en pantalla los caracteres recibidos. En este apartado se comentarán los procedimientos y funciones más interesantes, omitiendo la descripción de las constantes que aparezcan, pues éstas se refieren en su mayoría a valores de los registros del puerto serie, tema que fue tratado en el artículo del mes anterior en esta misma sección.

En el procedimiento *inicializa_puerto* (en el CD-ROM) se procede a la inicialización de los valores de los registros del puerto serie, pero antes de que esto ocurra se inicializan los punteros que indican el primer y el último carácter que queda pendiente por tratar en la cola circular. Puesto que se va a proceder a reemplazar la rutina de tratamiento de interrupciones original por una nueva, representada por el procedimiento *trata_interrupcion* (cuadro 4), se desactivan las interrupciones mientras que se procede a la inicialización, se obtiene la dirección donde se encuentra la anterior rutina de tratamiento de interrupción y se procede a indicar al programa que se incluya en la secuencia de procedimientos de salida el procedimiento restaurar (asignación a *exitproc* del cuadro 4). Este proceso se realiza para preservar la antigua rutina si por cualquier motivo se procede a interrumpir el programa, o bien, éste tiene una salida anormal. A continuación se procede a utilizar el procedimiento *setintvec*, que tiene como parámetros el número de interrupción y la dirección que contendrá el vector de interrupción correspondiente, que no será más que un procedimiento *interrupt* construido al efecto, trata_interrupcion. Una vez inicializado el vector de interrupción, se procede a establecer los valores de los registros del puerto serie, activando por último las interrupciones, pues ya hay una rutina que se encarga de tratarlas.

El procedimiento *trata_interrupcion*, presenta la declarativa *interrupt;* por tanto, antes de ejecutarse la CPU guardará los registros de la tarea que estaba ejecutando en ese instante y pasará a ejecutar las instrucciones de este procedimiento, en definitiva muy sencillo: en primer lugar activa las interrupciones mediante la instrucción *inline($FB)*, obteniendo a continuación la siguiente posición libre de la cola circular; en ella almacenará el carácter que se ha recibido en el puerto. Por último, se indica al controlador de interrupciones que se ha finalizado el tratamiento de la interrupción, a través del puerto que éste tiene asignado (valor de ctrl_int1). Por último, debido a la directiva que acompaña a este procedimiento, se procede a la carga de los valores de los registros que previamente se guardaron, procediendo a la ejecución de la tarea que se estaba ejecutando cuando fue interrumpida.

El procedimiento restaurar (cuadro 4) se encarga de desactivar las interrupciones hacia el puerto serie volviendo a establecer su valor original, que se encuentra almacenado en la variable anterior_vector. Hay que indicar que para este procedimiento se ha utilizado la directiva de compilación {$F+}, para permitir que dicho procedimiento sea "lejano", permitiendo un cambio del segmento de código.

Existen otra serie de procedimientos y funciones que permiten consultar la cola en busca de nuevos caracteres que han llegado y enviar caracteres al puerto serie; debido a su simplicidad no se comentará nada acerca de ellos.

Cuadro 4.

```
{$F+}

procedure restaurar;
 begin
  port[ier_com2]:= 0;
  port[ctrl_int1]:= port[ctrl_int1] or irq3;
  port[mcr_com2]:= 0;
  setintvec(numint_com2, anterior_vector)
 end;

{$F-}

procedure trata_interrupcion; interrupt;
 begin
  inline($FB); {activacion de las interrupciones}
  if cola.ultimo = tam_buffer {preparacion para recibir caracter}
   then
    cola.ultimo:= 0
   else
    inc(cola.ultimo);
  cola.contenido[cola.ultimo]:= char(port[rbr_com2]);
  port[ctrl_int2]:= $20 {fin de interrupcion al controlador de interrupciones}
 end;
```

SEÑALES, ALARMAS, TEMPORIZADORES Y ELEMENTOS AFINES

Fernando J. Echevarrieta

Como ya se justificó en el primer artículo de la serie, la misión fundamental de un S.O. es la definición de una máquina virtual software para la que sea "sencillo" trabajar. Esta máquina virtual software sigue una analogía con una máquina real hardware en la que el conjunto de instrucciones de máquina se corresponde con un conjunto de llamadas al sistema y las interrupciones o *traps* hardware se corresponden con el concepto de señal.

Uno de los mecanismos de control más interesantes que proporciona la máquina virtual UNIX es precisamente el de señal. En este S.O. es muy simple manejar y tratar señales y, del mismo modo que en una máquina física existe un controlador de interrupciones y se puede realizar un enmascaramiento de las mismas, en UNIX es posible definir manejadores de señales así como enmascararlas.

En el presente artículo se presentarán los mecanismos para el manejo de señales en UNIX y se expondrán aquellos factores a tener en cuenta a la hora de programar con ellas ya que, como eventos asíncronos que afectan a un proceso, el momento de su aparición es impredecible y, si no se tiene esto en cuenta, pueden producir efectos imprevisibles. El mes pasado, por ejemplo, se desarrolló una librería para el envío de mensajes delimitados sobre *sockets* TCP y, aunque en principio parecía ser sólida, aún hay factores a tener en cuenta para su correcto funcionamiento. En la segunda parte del artículo se retomará esta librería e, introduciéndola en un entorno de señales, se observará cómo su funcionamiento es incorrecto y se modificará para ser robusta ante éstas.

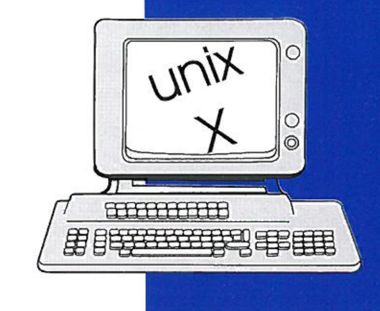
SEÑALES EN UNIX

Como se ha anticipado, una señal en UNIX es un evento software análogo a una interrupción. En la mayoría de los casos, este evento se genera de forma asíncrona, por lo que desde un proceso no es posible predecir cuándo se va a producir.

Cada señal UNIX se relaciona con tres parámetros :

- Un único número entero, único para cada señal y diferente del de todas las demás. Este número sirve para identificar de forma unívoca a la señal.
- Un nombre simbólico, relacionado de forma unívoca con el número entero identificador y cuyo único objetivo es identificar de forma más cómoda la señal a un programador humano.
- Una acción por defecto, que se ejecutará cada vez que un proceso reciba la señal. Esta acción por defecto en la mayoría de los casos puede ser cambiada.

Para hacer uso de las señales desde programa, habrá que incluir el fichero <signal.h> en las fuentes de los programas. En este fichero se define el nombre asociado a cada una de ellas. En la tabla 1 se puede observar una relación de identificadores numéricos y simbólicos, así como su acción por defecto. Como se puede observar, la mayoría de ellas provocan que el proceso aborte y, en algunos casos, genere un *core dump*, o volcado de memoria. Este volcado, que aparece en disco como un fichero de nombre *core*, servirá posteriormente para realizar una "autopsia" al proceso, permitiendo analizar el estado del mismo antes de su muerte y, de



Uno de los mecanismos más interesantes que proporciona UNIX es el manejo de señales. En el presente artículo se hará una exposición del empleo de las mismas en este sistema operativo y se presentarán algunos factores a tener en cuenta cuando se programa con señales, para lo que se tomará como ejemplo y se continuará la librería de comunicaciones desarrollada el mes pasado.

este modo, determinar las causas de la misma. En otras ocasiones, la recepción de la señal provoca únicamente la suspensión del proceso. En cualquier caso, será posible ordenar a un proceso que las ignore (la mayoría, no todas) o asignar a cada una una rutina de interrupción diferente de la que tenían establecida por defecto.

<unistd.h> y <signal.h> en nuestro programa. *SIG_IGN* es una función proporcionada por el sistema.

- Cuando se desee asignar un manejador a la señal distinto del que tiene por defecto, se puede indicar mediante:

signal(signum, manejador) ;

TABLA 1.

Nombre	num	Efecto	Significado.
SIGHUP	1	exit	hang up
SIGINT	2	exit	interrumpe
SIGQUIT	3	core dump	termina
SIGILL	4	core dump	instrucción ilegal
SIGTRAP	5	core dump	trace trap
SIGIOT	6	core dump	instrucción IOT
SIGEMT	7	core dump	instrucción EMT
SIGFPE	8	core dump	excepción pto flotante
SIGKILL	9	exit	Mata (no atrapable)
SIGBUS	10	core dump	error de bus
SIGSEGV	11	core dump	violación de segmento
SIGSYS	12	core dump	mal parámetro en llamada al sistema
SIGPIPE	13	exit	escritura en pipe sin lector
SIGALRM	14	exit	reloj de alarma
SIGTERM	15	exit	señal de terminación (software)
SIGURG	16	discard	condición de urgente en socket
SIGSTOP	17	suspen	stop (no atrapable)
SIGTSTP	18	suspen	stop (teclado)
SIGCONT	19	discard	continúa tras stop
SIGCHLD	20	discard	cambio en status del hijo
SIGTTIN	21	suspen	intento de lectura term. desde background
SIGTTOU	22	suspen	intento de escribir en term. desde backgrd.
SIGIO	23	discard	posible I/O en un descriptor
SIGXCPU	24	exit	tiempo límite de CPU excedido
SIGXFSZ	25	exit	espacio límite de fichero excedido
SIGVTALRM	26	exit	alarma de tiempo virtual
SIGPROF	27	exit	alarma del temporizador profiler

Algunas señales UNIX.

IGNORANDO Y ASIGNANDO MANEJADORES A LAS SEÑALES

Para definir el comportamiento de un proceso ante la aparición de una señal, se emplea la función *signal*, cuya definición es:

*void (*signal(int signum,(void *handler)(int)))(int) ;*

Aunque ésta es de las funciones que, a primera vista, asustan a aquellos que no somos especialmente hábiles con C, en la práctica su manejo es muy sencillo:

- En el caso más simple, en que se desea ignorar una señal, bastará con indicar:

signal(signum, SIG_IGN);

siendo *signum* el identificador numérico o simbólico de la misma. Para ello, habrá que incluir los ficheros

donde *manejador* debe ser una función declarada de la forma:

void manejador(int) {
...
}

En la práctica, si la función manejador se ha definido de otra forma, se genera un *Warning*, del que uno, si sabe lo que hace, puede escaparse "chapuceramente" mediante un *casting* del tipo:

*signal(signum,(void *)manejador) ;*

- Si, en determinado momento se desea restablecer el manejador que el sistema operativo asigna por defecto a la señal, se emplea otra función preconstruida:

signal(signum,SIG_DFL) ;

- Por último, puede que en determinado momento, se desee cambiar el manejador asociado a una señal para, posteriormente, restablecer el original. Sin embargo, nada garantiza que el manejador original sea el establecido por defecto por el sistema operativo, ya que podría haber sido cambiado por otro módulo del mismo programa, por ejemplo. Así pues, conviene almacenar de alguna forma el manejador asociado antes de la sustitución. Por ello es por lo que la función *signal* devuelve un valor de retorno que corresponde a este manejador. Este valor, como se deduce "claramente" de la definición de la función deberá ser declarado en el programa como:

*void (*manejador_antiguo)(int) ;*

tras lo cual se podrá teclear con tranquilidad, por ejemplo:

manejador_antiguo=signal(signum,manejador_nuevo) ;

ya que para restaurar el manejador antiguo bastará ahora, en otro punto del programa, indicar:

signal(signum,manejador_antiguo) ;

A pesar de todo, hay que tener en cuenta que el comportamiento ante algunas señales como SIGKILL (la señal "asesina", 9) o SIGSTOP, no podrá ser cambiado.

ENVÍO DE SEÑALES

UNIX proporciona una forma de enviar señales de forma "artificial", es decir, de generarlas por programa. En los primeros artículos de la serie, dedicados al manejo de este sistema operativo a nivel de usuario, se presentó el comando *kill* como una forma de "matar" procesos. Entonces, ya se adelantó que algunos procesos recalcitrantes se "negaban a morir" por lo que era necesario eliminarlos mediante la opción -9 : *Kill -9 pid, ó kill -9 %jobid.* En realidad, el comando *kill* sirve para enviar señales a los procesos y, aunque se utiliza generalmente para matarlos, de lo que toma su nombre, es posible emplearlo para enviar cualquier tipo de señal. Así, por defecto envía una señal de terminación SIGTERM, 15, ordinaria, que puede ser atrapada o ignorada. Mediante *kill -9* se envía la señal, SIGKILL, 9 (la "asesina") que no puede ser atrapada ni ignorada. Pero suele ser común también enviar otras señales como SIGHUP, mediante *kill -1 pid* ó *kill -HUP pid.* Una forma de obtener una



lista de señales disponibles es *teclear kill - ?* o *kill -l*

El comando *kill* se sustenta sobre una llamada al sistema homónima, que responde a la forma:
int kill(pid_t pid, int signum) ;
donde *pid* se corresponde con el identificador de proceso "destinatario" y *signum* es la señal a enviar.

DANDO LA VOZ DE ALARMA Y ECHÁNDOSE A DORMIR

Una señal muy cómoda de enviarse a uno mismo es SIGALRM, para cuya activación UNIX proporciona la función *alarm*, con la sintaxis:

long alarm(long segundos)

Si se indica 0 como tiempo en segundos, se desactivará cualquier alarma previamente instalada. El valor de retorno indica precisamente el número de segundos que quedaban para que se enviara esta señal o cero si no había ninguna activada.

Para pasar un proceso a estado *sleep*, se dispone de varias funciones. Así,

unsigned int sleep(unsigned int segundos) ;
que ya apareció el mes pasado, deja el proceso que la emplea en suspensión durante un intervalo de tiempo en segundos. Asimismo,

void usleep(unsigned long microsegundos) ;
hace lo propio con un intervalo expresado en microsegundos.

int pause(void) ;
deja un proceso en suspensión hasta que recibe una señal. Esta función siempre retorna un valor -1.

UN SENCILLO EJEMPLO

En el listado 1 aparece un ejemplo muy simple en el que se ignora una señal, *SIGINT*, que corresponde a la señal generada por la pulsación en el teclado de CTRL+C y por defecto interrumpe el proceso. Asimismo, se asigna un manejador, *repone*, a otra señal, *SIGALRM*. Esta señal, por defecto, se suele emplear como "despertador" de procesos, aunque en este caso se la empleará para activar el manejador, que se encarga de reponer el manejador original de la señal *SIGINT*, que se había reemplazado previamente. El envío de la señal se realiza mediante la función *alarm()*. Aquellos lectores que deseen probar el programa lo encontrarán en el subdirectorio *basicos* que se creará al extraer los ejemplos correspondientes al artículo (ver punto "SOFTWARE FACILITADO"). Se podrá comprobar que durante los 10 primeros segundos, el programa ignorará la pulsación de CTRL+C, y volverá a atenderla transcurrido este tiempo.

LISTADO 1.

```
#include <signal.h>
#include <unistd.h>

void repone(void)
{
  signal(SIGINT, SIG_DFL);
     printf("El manejador habitual ha sido
repuesto\n");
}

void main(void)
{
   printf("Durante los próximos 10 segundos la
señal SIGNINT será ignorada\n");
  signal(SIGINT, SIG_IGN);
  signal(SIGALRM,repone);
  alarm(10);
  while (1) {}
```

Ejemplo de uso de señales

TEMPORIZADORES

UNIX asigna a cada proceso del sistema un conjunto de tres temporizadores, cada uno de los cuales cuenta hacia atrás en un dominio temporal distinto, por lo que no interfieren entre sí. Cuando uno de estos temporizadores llega al final de la cuenta atrás, envía una señal al proceso y, eventualmente, puede comenzar una nueva cuenta atrás. Los dominios de actuación de estos temporizadores se pueden observar en la figura 1, y su funcionamiento se detalla a continuación :

ITIMER_REAL: Este temporizador decrementa en "tiempo real" y, cuando termina, envía una señal SIGALRM al proceso.

ITIMER_VIRTUAL: Este temporizador decrementa únicamente cuando el proceso se encuentra en ejecución y, al finalizar, le envía una señal SIGVALRM.

ITIMER_PROF: Este temporizador decrementa cuando la CPU se encuentra trabajando para el proceso, bien porque el proceso se está ejecutando o porque el sistema operativo está actuando para el proceso, por ejemplo, en respuesta a una llamada al sistema.

Mediante la diferencia entre los valores de *ITIMER_PROF* e *ITIMER_VIRTUAL* es posible determinar el tiempo que el *kernel* ha estado actuando para el proceso en cuestión.

Para acceder a estos temporizadores desde programa, será necesario realizar el siguiente "*include*":
#include <sys/time.h>

Para ajustar los parámetros de un temporizador, se emplea la función:
*setitimer(int temporizador, const struct itimerval *valor, struct itimerval *valor_anterior)*

En ella, *temporizador* indica a cuál de los temporizadores se refiere la función y se pueden emplear las constantes anteriormente indicadas: *ITI-*

DOMINIOS TEMPORALES CORRESPONDIENTES A LOS TEMPORIZADORES DE CADA PROCESO

ITIMER_REAL
ITIMER_PROF
ITIMER_VIRTUAL

PROCESO EJECUTÁNDOSE	KERNEL TRABAJANDO PARA EL PROCESO	OTROS

Tiempo de CPU

Figura 1: Dominios temporales de los temporizadores UNIX

MER_REAL, *ITIMER_VIRTUAL* e *ITIMER_PROF*. Dado que *ITIMER_REAL* genera una señal SIGALRM no conviene utilizar *setitimer* y *alarm* juntas para evitar interferencias. El parámetro *valor* es un puntero a una estructura *itimerval*, que se define de la siguiente forma:

```
struct itimerval {
  struct timeval it_interval ;
  struct timeval it_value ;
}
```

En esta estructura, *it_value* determina el valor inicial de la cuenta atrás del temporizador, es decir, el tiempo de espera desde la activación hasta la generación de la señal. Como se ha indicado anteriormente, cuando un temporizador finaliza su cuenta atrás, puede detenerse o comenzar una nueva cuenta atrás. Este comportamiento se determina mediante el parámetro *it_interval*. Si su valor es 0, el temporizador se desactivará, mientras que si es distinto de cero, comenzará una nueva cuenta atrás desde el valor *it_interval* cada vez que llegue a cero, es decir, producirá una señal periódica de periodo *it_interval*.

En resumen, el funcionamiento de un temporizador es, por tanto, el siguiente:

1. Inicia una cuenta atrás entre los valores *it_value* y 0.
2. Envía una señal al proceso.
3. Se ajusta a *it_interval*.

3a. Si (it_interval == 0) se desactiva.

3b. Si (it_interval != 0) inicia una nueva cuenta atrás desde *it_interval*.

Tanto *it_interval* como *it_value* son, a su vez, estructuras, del tipo *timeval*, que ya ha aparecido en otros artículos de la serie. No son ganas de complicar las cosas, simplemente es la forma más simple de indicar unidades temporales en segundos y microsegundos:

```
struct timeval {
  long tv_sec ;
  long tv_usec ;
}
```

donde, como cabe adivinar, *tv_sec* indica los segundos y *usec*, los microsegundos.

LISTADO 2.

```
#include <signal.h>
#include <sys/time.h>

void inoportuno()
{
        signal(SIGALRM,inoportuno);

        printf("Molesto?\n");
}

main()
{
                struct itimerval valor, viejo;

                valor.it_value.tv_sec =5;
                valor.it_value.tv_usec=0;

                valor.it_interval.tv_sec =1;
                valor.it_interval.tv_usec=0;

                inoportuno();
                setitimer(ITIMER_REAL,&valor,&viejo);

printf("Viejo=%d\n",viejo.it_value.tv_sec);
                while (1) {}
}
```

Ejemplo de uso de temporizadores.

Dado que se puede emplear la función *setitimer* en mitad de una cuenta atrás, el parámetro *valor_anterior* de la misma es rellenado por el sistema para indicar los valores previos del temporizador. También existe una función para leer los parámetros de un temporizador en lugar de ajustarlos. A saber,

*getitimer(int temporizador, struct itimerval *valor)*

En este caso, *it_value* indica el tiempo restante en la cuenta atrás desde que el temporizador fue iniciado o cero si no se encuentra habilitado, e *it_interval* indica el tiempo de "*reset*" del mismo.

OTRO SENCILLO EJEMPLO

En el listado 2 se puede observar otro ejemplo en el que se emplean temporizadores para generar una señal de alarma periódica. El temporizador *ITIMER_REAL* se programa para realizar una cuenta atrás de 5 segundos y ajustarse tras ello a un valor de 1, por lo que generará una alarma con un período de un segundo tras el intervalo inicial de 5 segundos. Más adelante en este mismo artículo se realizarán algunos comentarios sobre el funcionamiento del procedimiento inoportuno.

PROGRAMANDO EN UN MAR DE SEÑALES

Llegados a este punto, es necesario hacer notar al lector cómo la aparición imprevisible de una señal puede echar al traste lo que considerábamos un programa "robusto". Por ejemplo, el mes pasado se desarrollaba una librería con dos funciones para el envío de mensajes sobre TCP. En una primera versión, que funcionaba en condiciones favorables, se pudo comprobar cómo, si los mensajes llegaban con cierta velocidad, se recibían entremezclados ya que, en realidad, TCP transmite un flujo de información y no delimita PDUs de nivel superior. Así, se realizó una segunda versión con delimitación de mensajes. También se detectaron numerosos fallos de implementación, al no haber tenido en cuenta que un mensaje podría haber llegado sólo parcialmente a la hora de hacer una lectura sobre un *soc-*

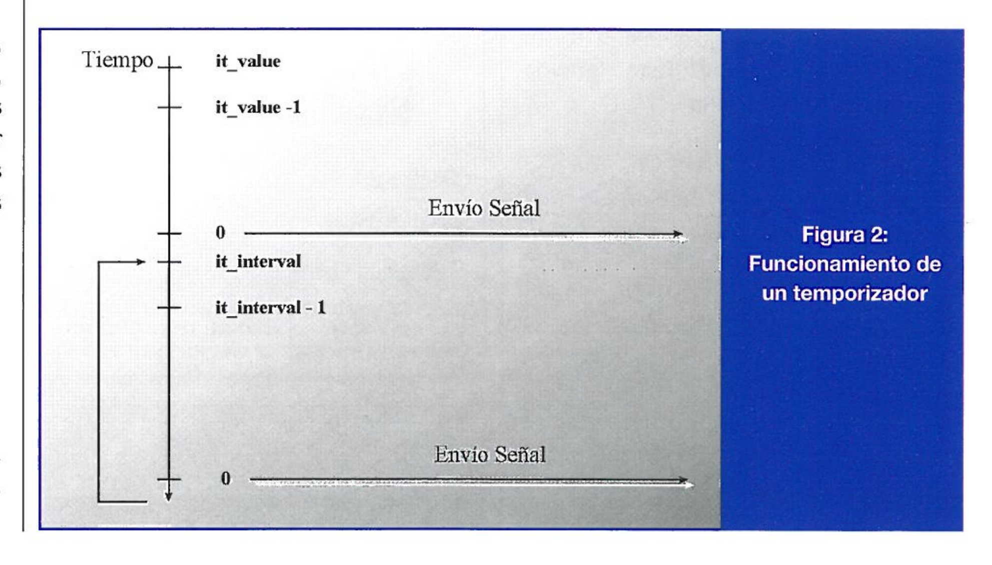


Figura 2: Funcionamiento de un temporizador

```
struct itimerval
    it_interval: tiempo de reset
        struct timeval {
          tv_sec:  tiempo en segundos
          tv_usec: tiempo en microsegundos
        }
    it_value: tiempo para expiración
        struct timeval {
          tv_sec:  tiempo en segundos
          tv_usec: tiempo en microsegundos
        }
```

Figura 3: Estructura itimerval

ket, por lo que se hacía necesario realizar sucesivas lecturas hasta asegurarse de que llegaba la totalidad de la información esperada. Asimismo, en casos más extremos, se hacía necesario considerar la posibilidad de que los *buffers* del sistema operativo se encontrasen llenos a la hora de realizar una escritura, por lo que también podría ser necesario el uso de sucesivas escrituras para la transmisión de un sólo mensaje. Con todo ello, se llegaba a la implementación de una librería en la que uno pudiera "depositar su confianza". Confianza que mal depositada quedaría si el programa se ejecutara en un entorno en el que pudieran aparecer señales. Por ello, se realizarán algunas pruebas introduciendo este nuevo elemento de ruido en el sistema. En el punto "SOFTWARE FACILITADO" al final del artículo se explica cómo compilar y lanzar los programas.

Con el fin de comprobar el comportamiento de la librería en un entorno en el que puedan aparecer señales, se ha optado por dotar al cliente *TCPcli* de un "inoportuno" procedimiento que, cada cierto tiempo *UPDATE_TIME*, interrumpe la ejecución del programa y pregunta si molesta. Para ello, simplemente se realiza un nuevo "*include*" en el programa:

```
#include <signal.h>
```

y se añade el procedimiento "inoportuno":

```
void inoportuno()
{
    signal(SIGALRM,inoportuno);
    alarm(UPDATE_TIME);

    printf("Molesto?\n");
}
```

Lo primero que hace el procedimiento es instalarse a sí mismo como manejador de la señal *SIGALRM*, mediante la función *signal*. Tras ello, indica al sistema operativo que debe lanzar una señal *SIGALRM* transcurridos *UPDATE_TIME* segundos y escribe en pantalla el mensaje "Molesto?". Al retornar, el programa continuará su ejecución, pero cuando reciba esta señal, lanzará el manejador adecuado, que muy hábilmente es el mismo procedimiento "inoportuno". Este procedimiento volverá a armar la señal y a escribir en pantalla. Y así, sucesivamente. De esta forma, se logra que aparezca en pantalla un mensaje con una periodicidad de *UPDATE_TIME* segundos generado por una interrupción.

En este punto, cabe realizar un comentario. Como se puede apreciar, el procedimiento no se "fía" del sistema y se instala a sí mismo como manejador de la señal cada vez que se ejecuta. Esto se debe a que, en unos sistemas operativos, una vez se ha ejecutado el procedimiento, se restaura el manejador por defecto mientras que, en otros, el nuevo manejador queda instalado hasta que no instale otro de forma explícita. Pero como la operación de instalar un manejador es *idempotente*, es decir, da igual si se instala una o n veces el mismo manejador, se garantiza un comportamiento definido realizándolo de esta manera.

Pues bien, si realizamos nuevas pruebas compilando con *make signal* podremos obtener dos tipos de resultados, ambos fallidos, como los que se aprecian en las figuras 4 y 5.

En la figura 4, se puede observar una primera ejecución en la que el procedimiento "inoportuno" hace su primera aparición, tras lo cual se produce un error de lectura y el programa aborta. Lo que ha sucedido es simple. La llamada al sistema *read*, bloqueante, ha sido interrumpida antes de haber sido capaz de leer nada. Por ello, ha retornado un valor menor que cero, que la función *sp_read* ha propagado. El programa lo ha interpretado como un error y ha abortado. Este problema se discutirá en el apartado siguiente. Para reproducir este comportamiento, lance *TCPcli* inmediatamente después de ejecutar *prueba*.

En el segundo caso, el procedimiento "inoportuno" también hace su aparición, pero solamente una vez (figura 5). Después, todo se ejecuta con normalidad y el procedimiento no vuelve a molestar. Para reproducir este comportamiento lance *TCPcli* un tiempo después de ejecutar *prueba*. Pero, ¿por qué no vuelve a molestar el procedimiento "inoportuno"? Lo que ha ocurrido es que cuando el *sp_read* ha tratado de leer por primera vez del *socket*, ya había información en él, por lo que ninguno de sus dos *read* ha necesitado esperar para recogerla. Por ello, su espera no ha sido interrumpida por ninguna señal (puede que sí haya habido espera, lo importante es que no ha habido interrupción de espera). El bucle ha continuado sin problemas hasta que se ha ejecutado la función *sleep()*. Esta "inocente" función, que parece la perfecta candidata para ser una llamada al sistema, ya que deja el proceso en suspensión, en realidad no lo es, sino que se trata de una función de librería que en algunos casos emplea la señal SIGALRM (dependiente de implementación) por lo que interfiere con ella, anulándola. Por ello, *sleep()* y *alarm()* no deben emplearse nunca juntas. La función *sleep()* puede ser fácilmente reemplazada por una simple llamada a *select*, de la forma:

```
void mi_sleep(int sec)
{
        struct timeval tv;
        tv.tv_sec=sec; tv.tv_usec=0;
     while(select(0, NULL, NULL, NULL,
&tv)< 0);
}
```

Con esta modificación ya no desaparece la actuación del procedimiento inoportuno, pero el programa aborta al ser interrumpida la librería de comunicaciones, como podrá comprobar el lector compilando el ejemplo con *make nosleep*. Se ha resuelto el problema de la interferencia entre funciones, pero la librería no es capaz de resistir interrupciones. Puede que en alguna ocasión el programa no aborte, será porque da la casualidad de que ninguna señal ha llegado mientras la librería de comunicaciones se encontraba en ejecución. Si esto le ocurre repita la prueba y fácilmente comprobará que el programa no es robusto.

Si se desea evitar el empleo de *alarm()*, en el listado 2, se vio cómo generar un mensaje periódico por medio del empleo de temporizadores en lugar de hacer uso de esta función.

GESTIÓN ATÓMICA DE MENSAJES DELIMITADOS

Como se comentaba en el apartado anterior, la culpa de que el programa aborte al producirse una interrupción en un *read*, la tiene el siguiente fragmento de código:

```
if ((c=sp_recv(s,buf)) > 0) {
...
} else if ( c == 0 ) {
...
} else {
  printf("Ha habido un error de lectura,
me muero\n");
  exit(1);
}
```

El lector puede pensar, en principio, que se podría salvar la situación sin necesidad de matar el programa. Sin embargo, esto no es así. Esto se debe a que no se está empleando directamente la llamada al sistema *read*, sino la función *sp_read*, que no se comporta de forma atómica.

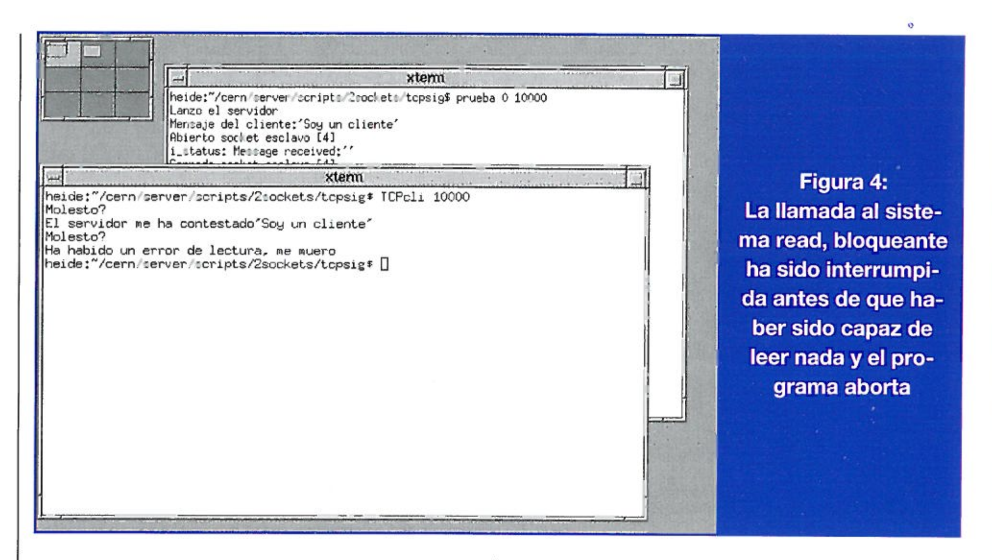


Figura 4: La llamada al sistema read, bloqueante ha sido interrumpida antes de que haber sido capaz de leer nada y el programa aborta

En efecto, toda llamada al sistema, como *read*, se comporta de forma atómica, es decir indivisible. Esto supone que se lleva a cabo completamente o no se lleva a cabo, pero si ha comenzado a leer del *socket*, no puede ser interrumpida por una señal procedente de una aplicación.

La llamada *read*, por defecto, se comporta de forma bloqueante. Cuando el programa realiza esta llamada al sistema, queda en suspensión hasta que algo se encuentre disponible en el descriptor de fichero (o de *socket*, lo que es indiferente). Si se produce una interrupción ANTES de que *read* haya sido capaz de leer nada, retorna con un valor de error -1 y le da a la variable *errno* (#include <errno.h>) el valor *EINTR*.

Por ello, si se estuviera empleando una llamada al sistema *read*, un código de retorno *EINTR* indicaría que se ha producido una interrupción antes de que fuera posible leer nada. Por ello, se podría volver a llamar a *read* como si nada hubiera pasado sin necesidad de salir del programa.

En el ejemplo, no se llama directamente a *read*, sino que se emplea una función *sp_recv* que se ha realizado previamente para la recepción de mensajes delimitados sobre TCP. El problema es que esta función no se encuentra bien implementada. En efecto, como se describió el mes pasado, *sp_read*, llama dos veces a *read*: una para leer las cabeceras de los mensajes y la otra para leer el cuerpo de los mismos y, en caso de producirse un error, retorna devolviendo el código de error con la esperanza de que la aplicación llamante resuelva el problema.

Esto es un grave error, ya que, por ello, no realiza la lectura del mensaje de forma atómica. Imagine el lector, por ejemplo, que la primera llamada a *read* de la función *sp_recv*, es interrumpida por una señal antes de que pueda leer una cabecera. La llamada a *read* devolverá un valor -1, y *sp_recv* retornará devolviendo un valor -1. La primera idea para proteger el programa sería propagar también el valor de *errno*. Sin embargo, si se hiciera esto, el programa principal, al percibir que *sp_recv* habría sido interrumpido por una señal, volvería a llamar a la función esperando que, como un *read*, se comportara de forma atómica. Sin embargo, al invocarse de nuevo, *sp_read* trataría de leer nuevamente una cabecera de mensaje del *socket*, cabecera que ya habría sido retirada en la anterior invocación. Al llamar a *read* por primera vez, en lugar de la cabecera del mensaje, estaría leyendo el cuerpo del mismo, con lo que se habría producido una desincronización de la comunicación. Gracias al empleo de un número mágico (ver artículo del mes anterior) se podría detectar esta desincronización, pero poco se podría hacer para resincronizar sin perder información.

En el caso del ejemplo, la solución más simple es programar bien la función *sp_recv* haciéndola atómica frente a las interrupciones. Así, bastaría con estudiar cada uno de los dos *read* que emplea para detectar si ha habido interrupción y, en caso afirmativo, realizarlo de nuevo. De esta forma, el progra-

ma de aplicación nunca sería consciente de que habría habido una interrupción. Así, en la lectura de la cabecera del mensaje, donde antes figuraba

```
retcode =
read(s,message,2*length+2);
if (retcode <= 0) return retcode;
```

ahora bastaría modificar para dejar

```
do {
  errno=0;
                    retcode =
read(s,message,2*length+2);
} while (errno==EINTR);
if (retcode <= 0) return retcode;
```

y en la lectura del cuerpo, donde antes figuraba:

```
do {
  errno=0;
                    retcode =
read(s,r_mesg.message+totRead,tryTo
read);
  if (retcode>= 0) {
    totRead+= retcode;
    tryToread-= retcode;
  }
  while ((retcode > 0) && (totRead <
msg_len));
}
if (retcode <= 0) return retcode;
```

bastaría cambiar la condición del bucle:

```
while (((retcode> 0) && (totRead<
msg_len)) || (errno==EINTR));
```

CONCLUSIÓN

A estos ejemplos hay que aplicar las conclusiones a las que se llegaba el mes pasado respecto a la robustez. Sobre todo hay que tener en cuenta que nuevamente se ha asumido que la cabecera del mensaje, por tener un tamaño reducido sí será leída en un solo *read*, y no se ha protegido del mismo modo que la lectura del cuerpo del mensaje. Pero eso era tema del artículo del mes pasado. En éste deberían haber quedado sentadas las técnicas básicas para el manejo de señales y temporizadores así como la forma de robustecer un programa frente a interrupciones indeseadas.

A la hora de programar funciones de librería puede ser conveniente que se comporten de forma atómica frente a las señales

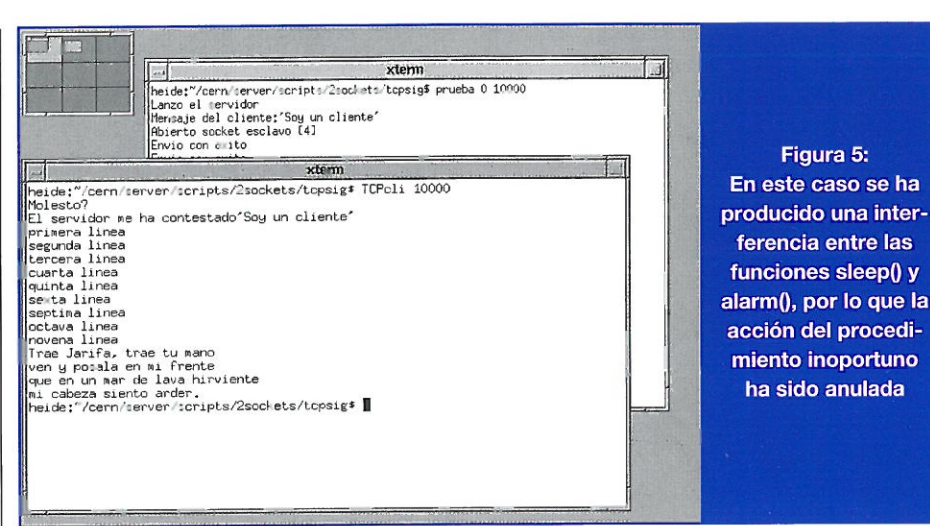


Figura 5:
En este caso se ha producido una interferencia entre las funciones sleep() y alarm(), por lo que la acción del procedimiento inoportuno ha sido anulada

SOFTWARE FACILITADO

Junto a este artículo se facilita el fichero *tcpsig.tgz* con programas ejemplo para Linux, para cuya extracción bastará teclear :
tar xvfz tcpsig.tgz
lo que generará un directorio *tcpsig* con un subdirectorio para ejemplo. Al igual que el mes pasado los ejemplos deben ser compilados y ejecutados DESDE EL DIRECTORIO PRINCIPAL haciendo uso de la utilidad *make* con :
make [signal | nosleep | atom]
La aplicación consta de una parte cliente, *TCPcli*, y una parte servidora, *TCPsvr*.
La sintaxis para realizar las pruebas será:
prueba <t> <puerto>
donde *t* es el tiempo en segundos entre mensaje y mensaje a enviar. El cliente, por su parte, deberá ser lanzado como
TCPcli <puerto>
debiendo ambos puertos coincidir.

BIBLIOGRAFÍA

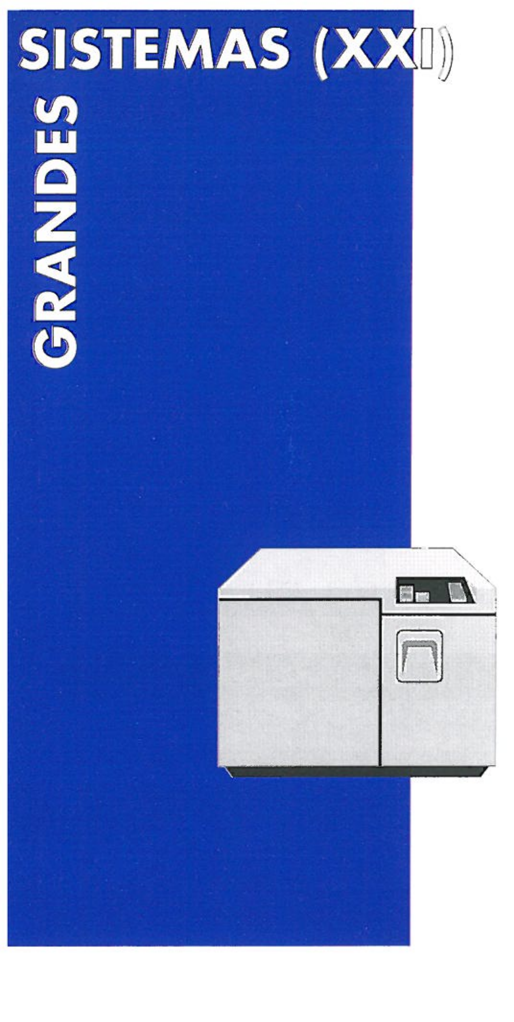
- Wang, Paul S., "*An Introduction to Berkeley UNIX*", Ed. Wadsworth, 1988
- Comer, Douglas E., "*Internetworking with TCP/IP*", volumes 1 & 3, De. Prentice Hall

Asimismo, los ejemplos básicos correspondientes a los listados 1 y 2 se encuentran en el subdirectorio *basicos* y podrán ser compilados tecleando *make ignora* y *make itimer* respectivamente.

CONTACTAR CON EL AUTOR

Para cualquier duda, consulta, sugerencia o crítica, relacionada con el artículo, se anima al lector a que se ponga en contacto con el autor a través de:
E-mail Internet: echeva@dit.upm.es
E-mail Compuserve: 100646,2456
WWW: http://highland.dit.upm.es:8000

Dada la elevada cantidad de mensajes recibida, el autor pide disculpas por no haber podido atender a todos los mensajes. Por ello, ruega que estos sean lo más concisos y directos que se pueda y a ser posible que cada cuestión se plantee en un mensaje independiente. Si en un plazo de 10 días no se recibe respuesta, podría ser conveniente enviar un nuevo mensaje. Asimismo solicita a los lectores que las cuestiones planteadas sean relativas a los artículos aparecidos en la revista. Si bien se tratará de contestar a todas, se dará preferencia a éstas.



CICS (III)

José María Peco

El pasado mes se inició un tema de gran aceptación en el mundo de los Grandes Sistemas, concretamente del monitor de teleproceso CICS, con el cual se pretende divulgar, a grandes rasgos, el funcionamiento interno de este tipo de productos.

El pasado mes, se comentaron los distintos conceptos básicos en los que se basa este monitor de teleproceso. El presente artículo describe los distintos componentes del CICS. No se fija como objetivo dar un curso, ya que ello llevaría muchos meses, sólo se pretende que el lector conozca los conceptos en los que se basa este entorno.

Los componentes que permiten a este monitor cumplir su objetivo son los siguientes:

- Programas de control para la Gestión del sistema.
- Servicios del sistema para la Gestión de tareas.
- Seguimiento del sistema: *Trace.*
- Fiabilidad del sistema.
- Soporte del sistema.
- Servicios a las aplicaciones: Traductores de comandos.
- Intercomunicación de sistemas.

GESTIÓN DEL SISTEMA

Este apartado engloba todos aquellos programas que facilitan la gestión y control del sistema.

1.- KC: Gestión de tareas.

Los programas encargados de realizar esta gestión se denominan DFHKCPxx (1). Estos módulos pueden considerarse los más importantes del sistema, ya que:

- Reciben el control directamente del sistema operativo.
- Deciden a qué tarea se debe ceder el control. Es decir asignan la prioridad de cada tarea en función del código de transacción, o del terminal, usuario, etc...
- Originan y finalizan tareas.
- Detectan y corrigen situaciones de saturación.
- Detectan y corrigen bucles de programas. En el caso de que un programa se meta en un bucle infinito.
- Etc.

2.- SC: Gestión de memoria.

Tal y como se dijo el mes pasado, el CICS, desde el punto de vista del S.O. se ejecuta como un programa más dentro del sistema. En consecuencia, para poder gestionar las infinitas transacciones (o programas de usuario) deberá gestionar también su propia memoria. Ésta, dentro de la región CICS, se encuentra dividida en dos tipos, el área dinámica, gestionada por el SCP (*Storage Control Program*) y el área estática, su propia área de memoria.

La figura 1 muestra cómo el CICS tiene su propio gestor de paginación, de modo que cuando detecta que hay memoria que no se utiliza, la envía al fichero de paginación.

3.- PC: Gestión de programas.

Los módulos que configuran (DFHPCPxx) este componente del CICS proporcionan dos tipos de servicios:

3.1.- Servicios para las aplicaciones.

Son los encargados de proporcionar el enlace entre los distintos módulos, planificar la ejecución de los módulos asignando las correspondientes *WORKINGS* para cada programa, tal y como se comentó el pasado mes al hablar de la reentrabilidad (CICS es cuasi-reentrante) y planificar la comunicación entre los módulos de usuario y las funciones de CICS.

3.2.- Servicios para el sistema.

Estos módulos son los encargados de realizar:

- La carga asíncrona de los módulos, pues como ya se dijo, los módulos de usuario no son ejecutables en sí mismos, dado que no tienen *Working* propia, y además son considerados como subprogramas del programa principal de CICS;
- El control de uso de módulos para saber cuántas tareas están usando un módulo en concreto, y así poder

decidir si puede descargar de memoria un módulo.

- La figura 2 muestra la relación entre el gestor de programas y el gestor de memoria dinámica, ya que antes de cargar un módulo, examina en la PPT si ya está en memoria, y si no es así, pregunta al SCP si hay espacio para cargarlo.

4.- TC: Gestión de terminales.

Esta gestión es realizada por los programas TCP (*Terminal control program*) y ZCP. Es la tarea de más alta prioridad, y necesita estar complementada con un método de acceso de telecomunicaciones (VTAM), ya que:

- Inicia las tareas solicitadas desde los terminales.
- Asigna espacio a las áreas de E/S de terminales.
- Intercepta las peticiones de E/S de los programas al terminal.
- Intercepta y recupera errores de terminal.

La figura 3 muestra de forma esquemática la comunicación entre el terminal y el *host*, cualquier cosa que se escriba en el terminal remoto del usuario queda reflejado en un *buffer* del VTAM, y es el CICS quien se encarga de examinar éstos, mediante el uso de determinados comandos para determinar si deben ser tratados por él.

Asimismo, para poder conseguir que cada mensaje llegue al terminal que ha iniciado la transacción, necesita apoyarse en dos tablas: las tablas de control de terminales del CICS y la tabla de recursos de comunicaciones del VTAM.

5.- FC: Gestión de ficheros.

La gestión de los ficheros es realizada por los módulos DFHFCPxx, siendo su función la de permitir el acceso a los ficheros o conjuntos de datos definidos para el CICS.

En este sentido cabe resaltar el hecho de que en MVS (2), para poder leer datos de un fichero, éste debe estar asignado al entorno del programa que quiere usarlo, es decir, del CICS. Por esta razón se puntualiza el hecho de que, en el procedimiento de arranque del monitor, o deben estar definidos" todos los ficheros que puedan tratarse desde este subsistema.

Por lo que se refiere a la seguridad de los datos, esta puede ser :

- Interna: Se define en la FDT (*File Definition Table*).
- Externa: Se realiza mediante el RACF, TOP SECRET o cualquier otro gestor de seguridad del sistema, el cual permite o deniega el acceso a cualquier fichero en función del nombre del usuario y del fichero.

Otro punto importante dentro de este apartado es el referido a la integridad de los datos, la cual se consigue en accesos concurrentes mediante control exclusivo por intervalo de control (3), no de registro lógico; y en los casos de destrucción lógica o física de la información, recuperando ésta desde los archivos de *log* que suelen mantenerse en estos grandes sistemas.

La información contenida en esta tabla tiene que ver con los ficheros que pueden ser accedidos desde CICS

Este monitor de teleproceso da soporte a todo tipo de ficheros, tanto VSAM como DAM y proporciona servicios de altas, bajas (KSDS-VSAM), modificaciones, lectura al azar y lectura secuencial (lee incluso hacia atrás secuencialmente, cosa que no se puede hacer sin CICS con COBOL).

En cuanto a las Bases de Datos ADABAS, DB2, etc., son los propios fabricantes los que proporcionan al CICS los módulos necesarios para poder establecer la comunicación con sus productos.

6.- Gestión de datos TS / TD.

CICS diferencia en su entorno dos tipos de datos, transitorios (TD) y temporales (TS).

Quizá el término "mensaje" sea mas explicativo, pero en este artículo se sigue la terminología usada por IBM con este producto.

Pues bien, con ambos tipos se puede guardar datos para ser procesados posteriormente, recoger y tratar mensajes que están esperando para ir a un terminal ocupado, crear datos para ser procesados por otra tarea (4) y guardar información para la recuperación después de que se haya producido un error.

6.1.- Gestión de datos transitorios (TD).

Este tipo de datos se refiere a aquella información que se almacena secuencialmente en unos destinos simbólicos llamados colas, encontrándose éstos definidos en la DCT (Tabla de Control de Destinos). Los nombres de estos destinos tienen una longitud de 4 octetos.

Dentro de este tipo de datos se pueden diferenciar otros dos tipos: datos Extrapartición y datos Intrapartición. Los datos Intrapartición son aquellos datos transitorios que se generan y se procesan dentro del CICS. Su gestión se realiza almacenándose temporalmente en el fichero de destinos intrapartición (VSAM ESDS) que es definido por el administrador del CICS. Actualmente tiende a ser sustituido por memoria temporal (TS).

Los usos más normales de los TD Intrapartición son:

- Difusión: Envío de mensajes a muchos terminales.
- Conmutación de mensajes: Un terminal desea enviar datos a otro.
- Coleccionar datos para su proceso posterior.

Los datos transitorios extrapartición se generan dentro del CICS y se procesan fuera del CICS; o bien, se generan fuera del CICS y se procesan dentro, siendo su uso más normal:

- Registrar datos de las transacciones (LOG).
- Generar estadísticas del CICS y mensajes de error.
- Recoger datos para ser procesados OFF-LINE.

En cualquier caso, la grabación de estos datos se realiza siempre sobre

dispositivos secuenciales. El LOG del CICS es una cola extrapartición, ya que en el caso de que se quiera seguir la pista de cualquier hecho registrado en él, la información contenida en ese fichero se trataría por un programa ajeno al CICS.

6.2.- Gestión de memoria temporal (TS).
Como muy bien define su nombre, la información que se gestiona se almacena temporalmente en memoria, por eso es volátil y no se pueden borrar físicamente, pues no existen como tales registros, aunque lo que sí se puede hacer es marcarlos. A diferencia de los datos transitorios, los datos temporales son guardados como registros (*items*), de longitud variable, de unos nombres simbólicos œnicos llamados ÒcolasÓ (8 octetos). Son colas secuenciales. No es necesario definir en ninguna tabla los nombres simbólicos de las colas TS. Sólo habría que definirlos en el caso de querer que fueran recuperables al arrancar el CICS.

Las colas TS se pueden recuperar más de una vez. Su *ítem*, a diferencia de las TD, puede leerse tantas veces como se quiera. Existe un único índice de lectura y otro de grabación a nivel CICS.

Las colas que dan soporte a estos datos pueden definirse en memoria principal, dentro de la región de CICS y en memoria auxiliar (fichero de colas TS del CICS). Es estándar y está definida en arranque.

TABLAS DE GESTIÓN DEL CICS

Se incluye a continuación una relación de las principales tablas usadas por el CICS así como de sus nombres cortos o identificadores usados. Además todas las tablas se ven afectadas también por un sufijo (dos caracteres al final del nombre) para facilitar su manejo al administrador. Ejemplo: dentro de DFHTCT00 : COPY TCTJMP01

Así, se puede anticipar que las características de cada terminal están definidas en cada una de las entradas de terminal de la TCT. El programa de control de terminales (TC) tiene la lista de los terminales que están enlazados al CICS/VS, así como los mandatos de sondeo que pueden ser ensayados para saber si un terminal ha enviado alguna petición. Esta lista incluye información de qué terminales están funcionando, y de si una transacción está siendo procesada en un determinado momento.

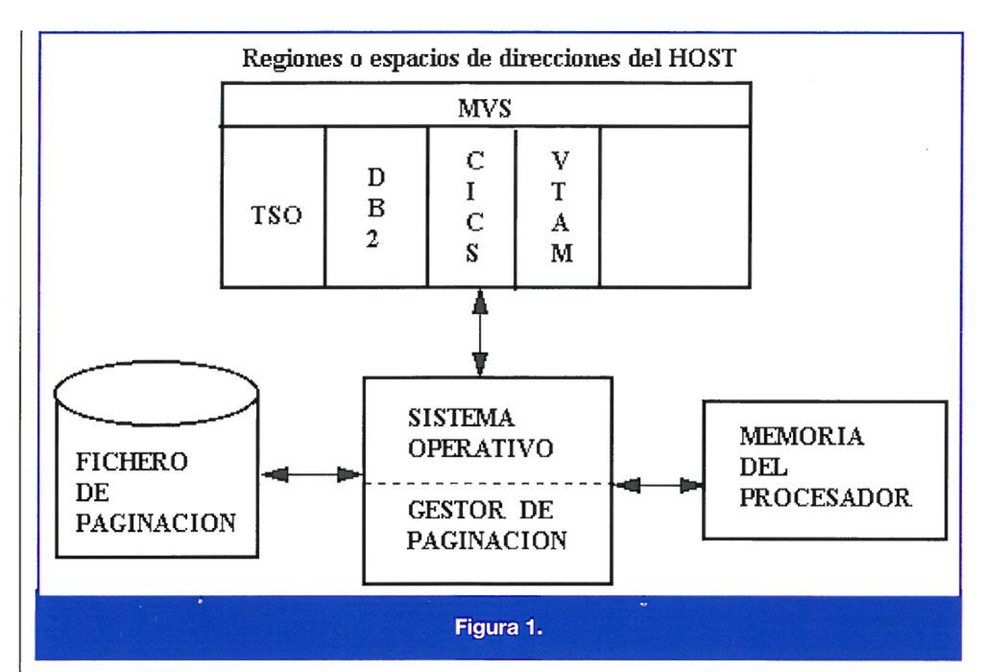


Figura 1.

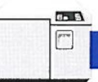
1.- SIT: tabla de Inicialización del CICS.
Si bien todas las tablas son fundamentales para el correcto funcionamiento del CICS, se podría decir que ésta es una de las mas importantes, ya que contiene información referente a toda la sesión CICS, tal como:

- Parámetros generales de CICS para el VTAM.
- Lenguajes en los que están escritas sus transacciones. (Se recuerda lo dicho anteriormente en el sentido de que debe generar las WORKINGs, razón por la que debe saber en qué lenguaje se escribió el programa asociado a la transacción). Por defecto vienen definidos los lenguajes COBOL y ASSEMBLER.
- Sufijos de las otras tablas DFHxxxSS (donde SS es el sufijo).
- Lista de recursos definidos en el fichero CSD.
- Numero máximo de tareas y tareas activas.
- Definición de cómo se va a realizar la seguridad (externa y/o interna).
- Longitud de la CWA (*Common Work Area*) que se comentará al hablar de los principales bloques de control.
- Parámetros de recuperación y rearranque.
- Parámetros asociados a las funciones del BMS (*Basic Mapping Support*). Este es un interfaz del que se hablará el próximo mes, y es el que permite independizar los datos tratados por la aplicación del tipo de terminal sobre el que se presentarán o capturarán los mismos. Ejemplo: BMS=(FULL,COLD,UNALIGN, DDS), con lo cual tiene activas todas las funciones básicas.

Normalmente todas las tablas están definidas en una misma librería o fichero PDS, y cada una de estas tablas son cada uno de los miembros de estas librerías. Así, esta tabla podría tener por DSN=CICS.JMP.TABLAS(DFHSIT01)

2.- SNT: Tabla de identificación de usuarios.
Esta tabla, que continuando con el ejemplo anterior, podría tener por DSN=CICS.JMP.TABLAS(DFHSNTxx), contiene la información referente a los usuarios que pueden identificarse para conectarse a CICS, tal como:

- Código de usuario.
- *Timeout* para el usuario, es decir, el tiempo que transcurrirá para que el sistema eche fuera al usuario si en ese tiempo no ha hecho uso del sistema.
- Seguridad interna de CICS.
- Seguridad externa del CICS (RACF).



En el caso de tener RACF la *signontable* (SNT), o tabla de firmas, puede tener sólo un usuario (genérico) ya que será el RACF quien verifique el AUTHID y asigne el perfil de autorización del usuario.

3.- TCT: Tabla de control de terminales.

Esta tabla contiene información referente a los terminales que pueden conectarse al CICS, tal como:

- Nombre del terminal para CICS (4 octetos). Ejemplo: TRMIDNT=ZU01
- Nombre del terminal para VTAM (8 octetos). Ejemplo: NETNAME=LP2404
- Tipo de terminal (pantalla, impresora, etc...)
- Tipo de acceso. Ejemplo: ACCMETH=VTAM
- Longitud de la TCTUA (*Terminal Control Table User Area*). Ejemplo: TIOAL=2000

Todas estas características pueden definirse en el fichero CEDA, aunque también se puede utilizar la autoinstalación aunque sólo si en el fichero CSD se ha definido un modelo de terminal que se ajuste a los parámetros de sesión que le pasa el VTAM. De esta forma, crea la TCT correspondiente a ese terminal. En este caso, los parámetros tienen que ser siempre los del VTAM.

4.- PCT: Tabla de control de transacciones.

Esta tabla, como indica su nombre, contiene información referente a las transacciones que pueden ejecutarse en CICS, tal como:

- Código de transacción: debe ser de 4 caracteres. Ejemplo TRANSID=JMP1. Este nombre, que es dado siguiendo la nomenclatura de la instalación, es el nombre con el que el usuario final le indica al sistema qué es lo que desea realizar: obtener el saldo, listar un programa, generar un cuadre, etc. CICS se reserva todos los nombres que empiecen por "C" para nombres de las transacciones propias del CICS (ejemplo: CEDA).
- Nombre del primer programa que debe ejecutarse cuando se invoque esta transacción. Ejemplo: PROGRAM=SAPJMP1P.
- Longitud de la TWA (*Task WorK Area*): además de la TCA asignada a cada tarea, en esta tabla podemos hacer que a cada transacción específica se le aumente la TCA en la TWA para pasar datos, etc.

Toda esta información también puede definirse en el fichero CEDA. La diferencia o ventaja entre utilizar este fichero o incluir estas definiciones en las tablas de inicialización del sistema estriba en el hecho de que usando este fichero las definiciones pueden usarse inmediatamente, mientras que en el caso de utilizarlas en las tablas de inicio, para que sean efectivas las modificaciones hay que cerrar y rearrancar todo el CICS.

El fichero de LOG es el encargado de grabar todas y cada una de las acciones realizadas por el CICS

5.- PPT: Tabla de control de programas.

Esta tabla es usada por el programa controlador de programas (PCP) para conocer la información asociada a los distintos módulos que pueden ejecutarse en CICS, tales como el nombre del programa, el lenguaje o la residencia, si va a ser residente o no, ya que si se especifica que va a ser residente, le cargará en la zona del área estática, y no le sacará de ella aunque necesite memoria para cargar otro programa.

También puede definirse esta información en el fichero CEDA. La información de una entrada de esta tabla, además de la información anterior, contiene un campo denominado "Cuenta de uso" para saber cuántos usuarios están usando este programa, pues se incrementa en 1 cuando una tarea quiere hacer uso de este programa, y se disminuye en 1 cuando termina.

Se recomienda ejecutar la transacción CEMT opción PR para consultar esta tabla.

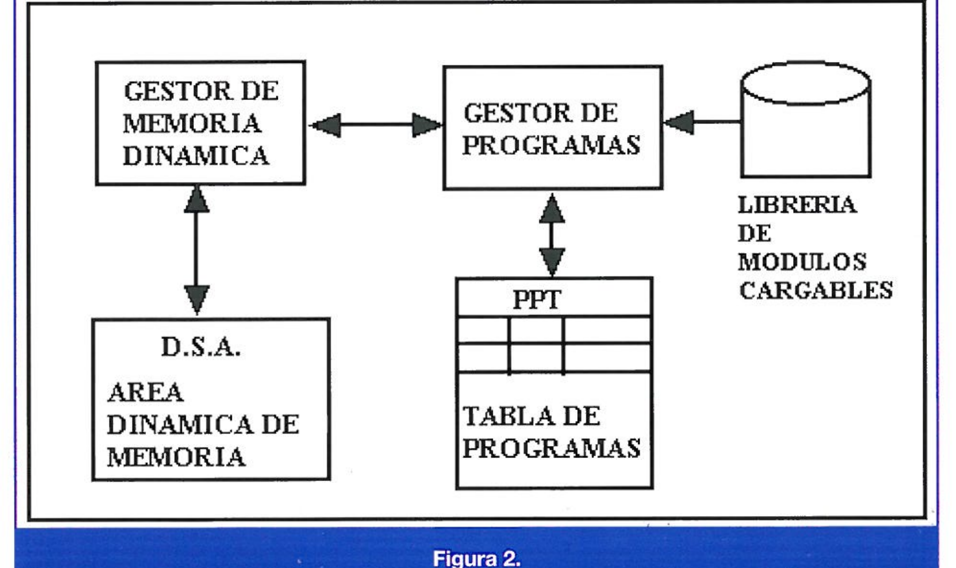


Figura 2.

6.- FCT: Tabla de control de ficheros.

La información contenida en esta tabla tiene que ver con los ficheros que pueden ser accedidos desde CICS. En consecuencia contiene entre otras la siguiente información:

- Nombre de la *Ddname* (o nombre corto con el que conoce el programa a dicho fichero). Normalmente el procedimiento de inicio es el que proporciona la relación entre la *Ddname* y el *Dsname* o nombre físico del fichero.
- Tipo de organización del fichero (VSAM, DAM, ...). Ejemplo ACCMETH=VSAM.
- Tipo de acceso: Lectura, altas,

bajas, modificaciones, lectura secuencial.

- Recurso protegido (para controlar en actualización). Ejemplo: SERV-REQ=(UPDATE,ADD,BROWSE,DELETE).
- *Buffers* que se reservan para el tratamiento del fichero, teniendo cada *buffer* el tama–o del intervalo de control.
- *Pool* de *buffers*: Para el caso de que no se use la opción anterior, le decimos que use un *Pool* de *buffers* de uso común entre todos los ficheros. Este Pool tendrá buffers de 8,4 y 2 K, por ejemplo, y usará el del tama–o correspondiente al tama–o del intervalo de control.

7.- DCT: Tabla de control de destinos.
Esta tabla contiene información referente a las colas de datos transitorios (Intrapartición y extrapartición) como el nombre de la cola (sólo para las TD), el nombre externo de las colas extrapartición (DDNAME) por ejemplo DSCNAME=DFHSTN, y las características físicas de las colas extrapartición, por ejemplo: RECFORM=VARBLK.

8.- JCT: Tabla de control de diarios.
En esta tabla se basa CICS para almacenar la información que permitirá realizar funciones de recuperación. El fichero de LOG es el encargado de grabar todas y cada una de las acciones realizadas por el CICS. Se corresponde con la entrada 1 de esta tabla. El resto de entradas se reservan para los ficheros JOURNAL. Estos ficheros contienen distinta información y no es obligatorio su uso. Así, se puede definir que guarde, en actualizaciones, la imagen del registro a modificar, antes y después de la actualización.

9.- PLT: Tabla de lista de Programas.
Existen dos listas, una con la lista de programas que se ejecutarán antes de ceder el control al TCP (Terminal Control Program), es decir, programas a ejecutar en el inicio, y una segunda lista que contiene la lista de programas que se ejecutarán al cerrar la sesión CICS.

PRINCIPALES BLOQUES DE CONTROL EN CICS

Si importantes son los programas que gestionan el propio monitor, y las tablas en las que se basa, también son importantes los bloques de control, buffers o mensajes en los que se basa el propio funcionamiento de este monitor de teleproceso.

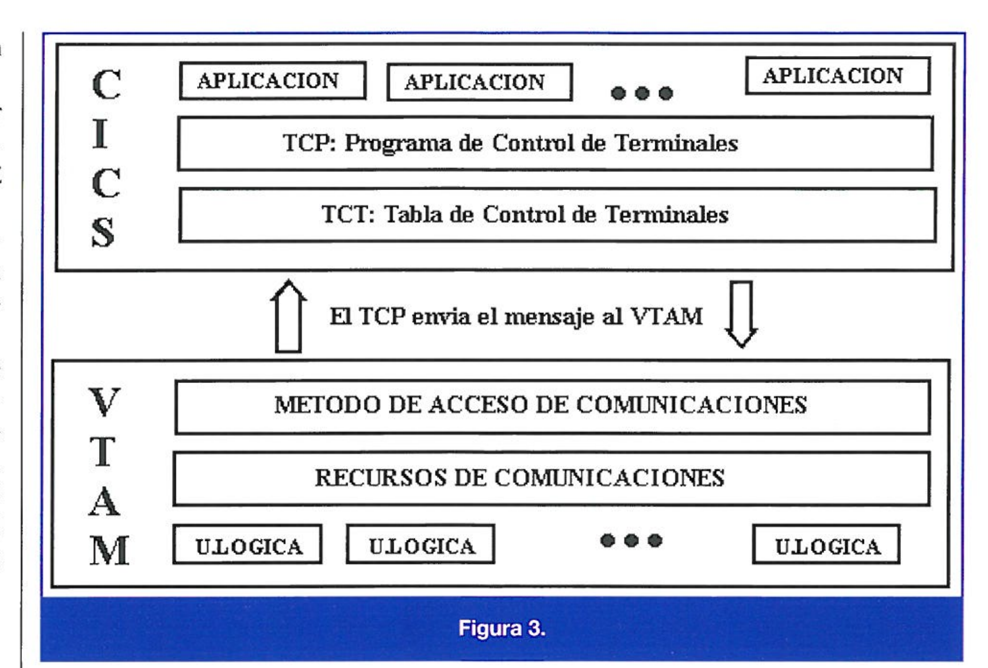


Figura 3.

RESUMEN DEL FUNCIONAMIENTO

Las características de cada terminal se encuentran definidas en cada una de las entradas de terminal de la TCT. El programa de control de terminales (TC), investigando esta lista, conoce cuáles son los terminales que están enlazados al CICS/VS así como los mandatos de sondeo que pueden ser ensayados para detectar si han enviado algún mensaje para ser procesado.

Como resultado de este sondeo, si detecta que un terminal tiene datos que enviar, el TCP obtiene un área de E/S (TIOA) e intenta leer los datos enviados desde ese terminal. Esta información leída es muy probable que contenga, entre otras cosas, el código de la transacción que se debe ejecutar. En consecuencia, antes de que se inicie una tarea, se ejecuta una comprobación previa para ver si existe esa transacción en la TCP, si el usuario está autorizado, etc...

Suponiendo que todo sea correcto, el controlador de tareas, KC, crea una nueva tarea de acuerdo con el identificador de la transacción para procesar los datos recibidos. Para realizar esto, crea varias áreas o buffers, y los apuntadores necesarios para encadenarlos en las distintas colas. Termina cediendo el control al PC (program Control) o controlador de programas, para que se inicie la ejecución de la transacción como una nueva tarea dentro del CICS. El PC usa la tabla PPT (de control de programas) para buscar la posición del programa en la memoria virtual o en el disco. Esta tabla contiene la información necesaria para el control de los programas, tales como lenguaje fuente, a fin de que se puedan realizar los enlaces correctamente. En el caso de que el programa no estuviera en memoria virtual, el control de programas se encargará de que el módulo se cargue antes de cederle el control.

NOTAS

(1) Se recuerda que todos los módulos que pertenecen a este producto comienzan por DFH
(2) MVS es el S.O. para grandes sistemas de IBM.
(3) El concepto de intervalo de control se trató en el tema de ficheros en Solo Programadores (Marzo-abril-mayoi96).
(4) Tarea: unidad de proceso interno para realizar una transacción.

PROXIMO MES

Una vez más, razones de espacio impiden completar el tema así que se deja para el próximo mes una visión completa del funcionamiento de este monitor y de algunas particularidades del mismo, a fin de facilitar su manejo.

REDES LOCALES NETWARE DE NOVELL

María Jesús Recio

En los dos artículos anteriores se habló de los protocolos IPX/ SPX, así como de la instalación de una red Netware. Es conveniente recordar que el servidor de Netware se puede instalar sobre una partición de un disco duro de un PC. Al instalar el servidor se genera un directorio en la partición DOS del disco, llamado por defecto NWSERVER, que contiene, entre otros muchos programas de Netware, un programa ejecutable llamado SERVER.EXE que se encarga de cargar el servidor de ficheros de Netware sobre el ordenador.

Recordar también que el servidor de ficheros de Netware es dedicado; esto significa que cuando la máquina que lo soporta está ejecutando dicho servidor "no sirve para otra cosa". No obstante es posible cargar algunas utilidades incorporadas por Netware dentro del propio servidor de ficheros, las cuales van a permitir gestionar la red y realizar modificaciones en la instalación. Cabe destacar como utilidades: *Monitor* e *Install*.

Monitor es una herramienta que permite, entre otras muchas cosas, hacer seguimientos de conexiones, realizar estadísticas del rendimiento del servidor, comprobar los ficheros abiertos por cada conexión, etc.

Install es una herramienta que va a permitir fundamentalmente la instalación de nuevos módulos en el servidor, así como la modificación de algunos ficheros de configuración del sistema.

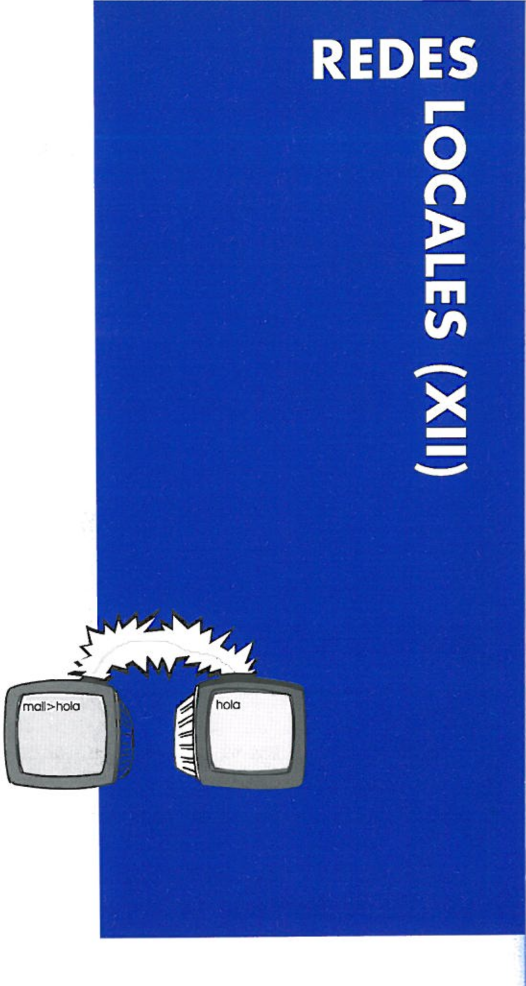
CONVIVENCIA DE NETWARE DE NOVELL Y WINDOWS

Ya se ha dicho que, cuando se instala una red Netware de Novell, la única constancia evidente de la conexión en red que tienen las máquinas clientes es la visualización del servidor, siempre y cuando se haya establecido correctamente la conexión con él. Recordar que, cuando se carga el cliente Netware en una máquina, automáticamente se visualiza una mínima parte del disco duro del servidor; es decir, la conexión se ha establecido; pero hasta que el usuario no se identifique con un *login* y su *password* asociada no podrá trabajar con el servidor.

Netware de Novell no está diseñado para implementar una red "peer to peer"

Visto el párrafo anterior, el lector debe llegar a la conclusión de que en una red gobernada por Netware las máquinas clientes ven únicamente al servidor, pero no pueden verse entre ellas.

Esto no quiere decir que un usuario no pueda comunicarse con otro usuario de la red, o pasarle información. Netware ofrece un sistema de mensajería que permite el traspaso de mensajes entre usuarios conectados al servidor de ficheros. El intercambio de información es un poco más complicado en Netware. Para traspasar información desde una máquina cliente a otra, es



Netware de Novell es un sistema operativo de red que funciona sobre MSDOS, Windows, y sobre otras arquitecturas. Ofrece la posibilidad de centralizar información en un servidor de ficheros, a la vez que centraliza el acceso a dispositivos físicos como impresoras, módems, fax, discos duros, etc.

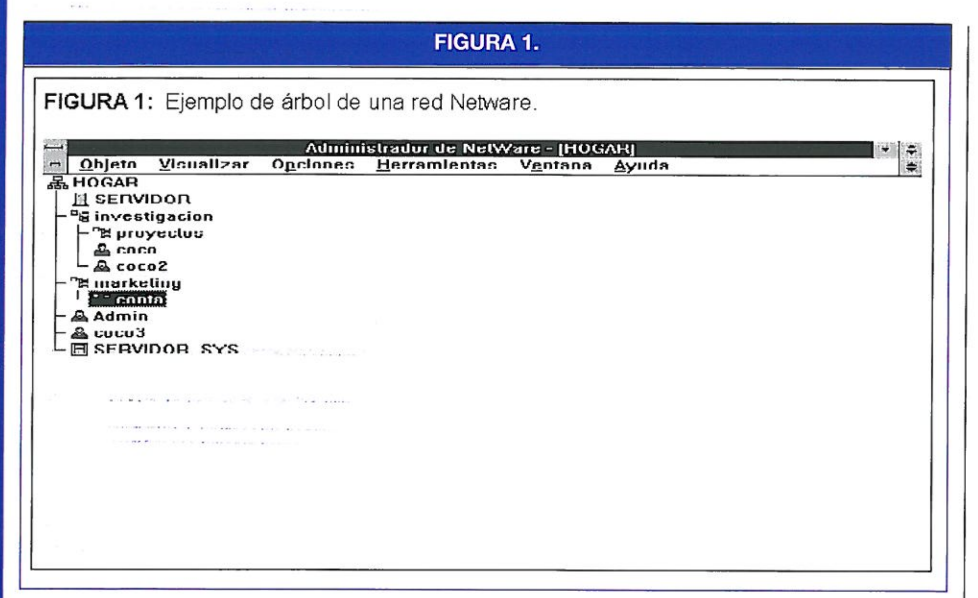


FIGURA 1.

FIGURA 1: Ejemplo de árbol de una red Netware.

necesario utilizar el servidor de ficheros de Netware como intermediario. Pero no basta con esto; además, los usuarios que quieran intercambiar información deben tener acceso a un directorio común que funcione como una especie de bandeja que permita dicho intercambio. Esta bandeja funcionaría de la siguiente forma:

Cuando un usuario quiera pasar un fichero a otro usuario, lo que debe hacer es copiar dicho fichero en un directorio bandeja al que el usuario destino tenga acceso (permisos de acceso a dicho directorio); una vez copiado en dicho directorio, el usuario destino deberá copiarlo desde este directorio a un directorio de su máquina local.

Como puede verse, Netware de Novell no está diseñado para implementar una red "*peer to peer*", sino más bien una arquitectura cliente-servidor.

Este tipo de redes se instalan por dos motivos principales:

- Para centralizar el acceso a cierto tipo de información que debe ser común a todos los usuarios; por ejemplo bases de datos.
- Para ahorrar espacio copiando una única vez las aplicaciones que van a ser utilizadas en la red (en el servidor), en lugar de realizar una copia local en cada máquina cliente de la red.

Pese a las ventajas que supone tener una red con este sistema de trabajo, a veces se hace necesario el poder compartir recursos locales, es decir, recursos localizados en una máquina cliente. Para realizar este tipo de operaciones se crean las redes "*peer to peer*", que son redes gobernadas con sistemas como Windows trabajo en grupo y Windows 95.

Las últimas versiones de Netware están diseñadas para manejar incluso WAN

Netware de Novell, así como Windows trabajo en grupo y Windows 95, ofrecen la posibilidad de convivir juntos, de modo que puedan aprovecharse las ventajas de cada uno de los dos sistemas de trabajo. Se puede instalar una red Netware, con su servidor de ficheros y sus clientes, y sobre ella instalar una red *Microsoft*, para poder compartir recursos locales a nuestra máquina, como pueden ser discos, impresoras, etc. De esta forma es posible trabajar tanto con el servidor como con cualquiera de las máquinas de la red. Obviamente, la naturaleza de lo que se puede realizar con cada una de las dos redes establecidas es diferente, puesto que están pensadas para trabajos diferentes.

ESTRUCTURACIÓN DE OBJETOS EN NETWARE

Netware ofrece, a partir de su versión 4.0, una estructuración de objetos muy potente. Se basa en definir cada elemento de la red -ordenador, servidor, volumen, impresora-, como si se tratase de objetos. Pero éstos no son los únicos objetos que se pueden encontrar. Los usuarios, los grupos de trabajo, los perfiles de usuario, etc, son otros objetos representados en el árbol.

Además permite abstraerse de la localización que ocupan dichos objetos, pudiendo ver su estructura jerárquica como un árbol, jerarquizado dependiendo no de la localización física del objeto, es decir, de su ubicación en la red, sino de su conexión y relación con el resto de los objetos de la red.

De esta forma consigue que una red, independientemente del tamaño y de la distancia que separa las máquinas que la componen, sea vista como una estructura única. Las primeras versiones de Netware sólo ofrecían la posibilidad de gestionar redes de área local; sin embargo, las últimas versiones están diseñadas para manejar incluso WAN.

Netware posee la capacidad de distribuir sus objetos en varios niveles de jerarquización, a los que llama: organización y unidades organizativas. Dentro de cada uno de ellos se pueden incluir otros objetos. La figura 1 muestra un ejemplo de una estructura jerárquica de un árbol visto desde *Nwadmin* (utilidad que ofrece Netware para la administración de objetos de la red, basada en entorno Windows).

En esta figura puede observarse que el objeto organización se llama HOGAR y es el que representa toda la red; de él cuelgan objetos tales como un servidor de ficheros, llamado SERVIDOR, un volumen del servidor (volumen SYS del servidor SERVIDOR), dos unidades organizativas: investigación y márketing, y dos usuarios: *admin* (administrador de la red) y *coco3*. Las unidades organizativas, además, contienen otros objetos, que son otras unidades organizativas y usuarios.

Estudiando esta figura puede llegarse a la conclusión de que existen dos tipos de objetos: objetos contenedores

y objetos terminales. Los objetos contenedores no representan más que una forma de organización del resto de los objetos; es, por asemejarlo a algo que el lector ya conoce ampliamente, como los directorios de un sistema de ficheros. Los objetos terminales, denominados en algunos libros como objetos hojas, son objetos finales que representan algo "real"; por seguir con el mismo ejemplo, son como los ficheros. La tabla 1 muestra todos los tipos de objetos disponibles en Netware 4.1.

Con esta forma de representación, una máquina localizada en cualquier sucursal de una empresa (es decir, en distintas ubicaciones físicas), se verá no como algo independiente, sino como algo perteneciente a una unidad organizativa. Por ejemplo, si el departamento de márketing está formado por objetos repartidos por sucursales localizadas en diferentes países, en el árbol se verán como objetos contenidos directamente en la unidad organizativa márketing, independientemente de su localización física.

Esta forma de unir todos los elementos de la red ofrece muchas facilidades al administrador para el manejo y el mantenimiento de dicha red.

Debido a esta jerarquización, a la hora de hacer referencia a un objeto, lo mismo que cuando se hace referencia a un fichero en una estructura de directorios, es necesario indicar su localización dentro del árbol. Para ello se deben seguir las siguientes reglas de sintaxis:

- Cada contenedor por el que se pasa para crear el camino al objeto está separado del anterior por el carácter punto (.).

- La ruta se empieza especificando el nombre del objeto y a continuación los objetos contenedores (al revés de como se hace cuando se especifica el *path* de un fichero).

- No es necesario especificar el nombre completo (ruta absoluta), sino que se puede tener en cuenta la localización en el momento de especificar el objeto, y crear la ruta terminando en este punto.

A la ubicación de un objeto se le llama contexto.

FICHEROS DE CONFIGURACIÓN DE UN SERVIDOR DE FICHEROS NETWARE

AUTOEXEC.CFN

En el artículo anterior se introdujo la misión que este fichero tenía en el servidor de Netware, y cuáles eran sus funciones. A modo de recordatorio puede decirse que este fichero se encuentra ubicado en el directorio SYS:SYSTEM y que su ejecución se lleva a cabo después de la carga del sistema, es decir, después de haber montado el volumen del sistema (recuérdese que el volumen SYS se monta automáticamente cuando el controlador de disco correspondiente se carga durante la ejecución del fichero *startup.cfn*) y en él se realizan operaciones como son la carga de módulos, definición de la configuración del sistema, montaje de sistemas de ficheros, almacenamiento de datos referentes a la red, entre los que se cuentan direcciones IPX, nombre del servidor, controladores de tarjetas, etc, es decir, todo aquello necesario para completar el proceso de arranque una vez ejecutados *server.exe* y *startup.ncf.* La tabla 3 muestra los comandos que deben introducirse en este fichero para la realización de otras operaciones no descritas en el artículo anterior. La tabla 4 plasma la lista de tipos de parámetros asociados a la orden *set.*

Para que dos sistemas se comuniquen directamente, deben utilizar el mismo protocolo para cada uno de los tres primeros niveles

STARTUP.CFN

Se trata de un fichero localizado en la partición DOS y que se ejecuta cuando se está cargando el servidor de ficheros de Netware; contiene órdenes que permitan inicializar el sistema: controladores de discos, definición de variables de entorno, etc. Es necesario destacar que la definición de variables de entorno con la orden *set* se puede realizar tanto en este fichero como en el fichero *autoexec.cfn.*

LA UTILIDAD INETCFG

Se trata de una herramienta de la consola que permite configurar y personalizar los protocolos IPX, IP y *Appletalk*, simplificando, por tanto, el proceso de configuración de redes de área local. Puede utilizarse mientras el servidor está funcionando.

Ya se ha visto que para cargar los protocolos de red basta con añadir ciertas líneas en los ficheros de arranque, tanto para la máquina servidora como para la cliente. Sin embargo, si se quiere configurar con más detalle estos protocolos es conveniente utilizar esta herramienta. Para cargarla únicamente es necesario escribir :

load inetcfg

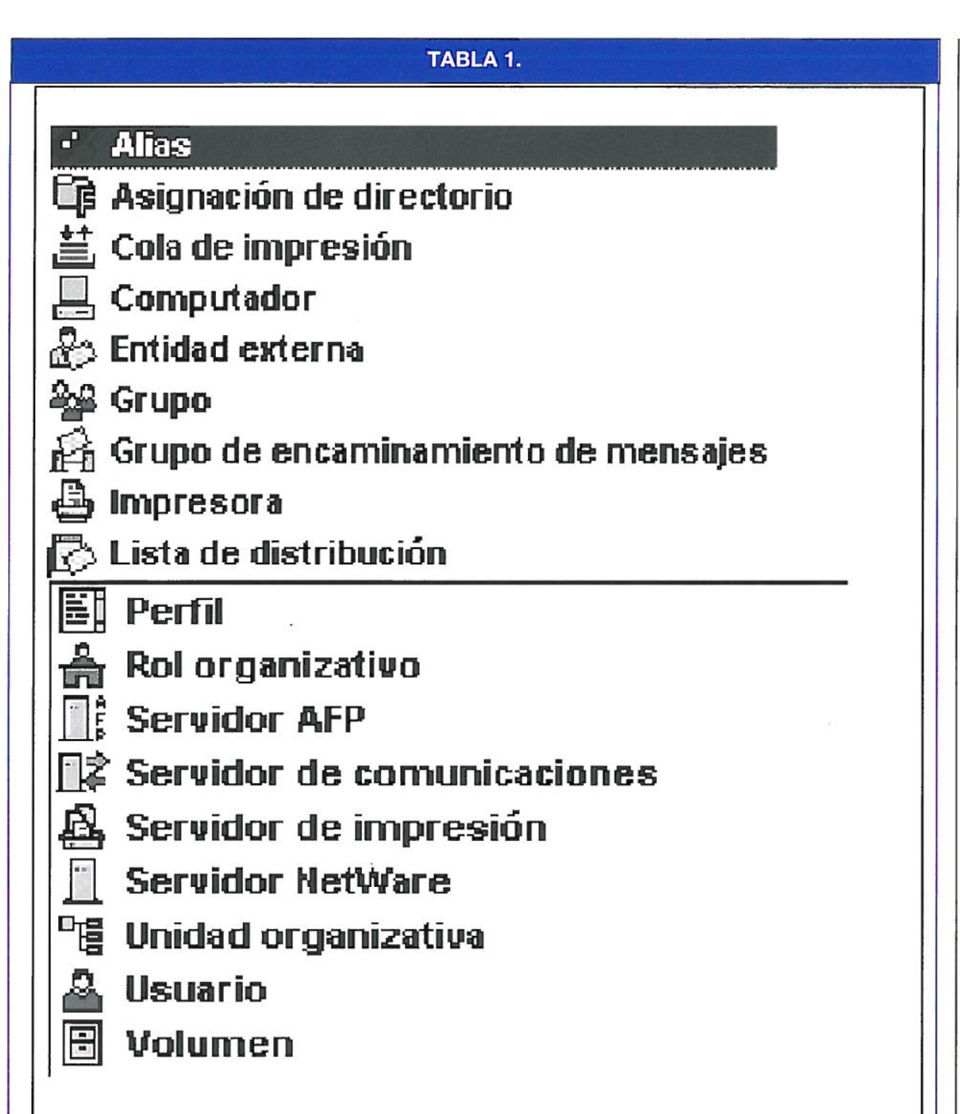
Al instante aparecerá un menú en la pantalla con las siguientes opciones:

- Tarjetas.
- Interfaz de red
- Directorio de llamada WAN.
- Protocolos.
- Asociaciones.
- Configuración de gestión.
- Ver configuración.

A grosso modo, se puede decir que estas opciones van a permitir desde seleccionar y configurar tarjetas de red, *drivers*, direcciones, IRQ's, hasta configurar los parámetros de los protocolos utilizados en la red, visualizar información de configuración, etc.

Inetcfg registra información en varios ficheros de configuración de Netware (.CFG), localizados en SYS :\ETC. Algunos de estos ficheros se encuen-

TABLA 1.


Alias
Asignación de directorio
Cola de impresión
Computador
Entidad externa
Grupo
Grupo de encaminamiento de mensajes
Impresora
Lista de distribución
Perfil
Rol organizativo
Servidor AFP
Servidor de comunicaciones
Servidor de impresión
Servidor NetWare
Unidad organizativa
Usuario
Volumen

tran presentes al inicio del sistema; otros, los que crea la propia utilidad, dependerán de cómo y qué configuren. Los principales de estos ficheros son :

aurp.cfg.
tcpip.cfg.
ipxspx.cfg.
nlsp.cfg.
netinfo.cfg.

Una vez realizados cambios de configuración, se actualizan automáticamente las bases de datos del sistema y se crean los comandos LOAD y BIND apropiados en el fichero de configuración correspondiente.

Una vez realizadas modificaciones con esta herramienta, los cambios en la configuración, en la mayoría de los casos (habilitación/inhabilitación de una tarjeta, modificación en los parámetros de la tarjeta, configuración de un nuevo protocolo y modificación de alguno de los parámetros de éstos, asociación de un protocolo a un interfaz de red) se activarán después de apagar el sistema y volverlo a arrancar ; otros, como los cambios de configuración de los filtros, se activan de manera automática.

GESTIÓN DE PROTOCOLOS EN NETWARE DE NOVELL

El *software* de protocolo de Netware 4.1. ofrece servicios muy completos; por ejemplo, posibilita la comunicación entre sistemas que están en segmentos de LAN diferentes, independientemente de la topología de cada segmento. Soporta, como ya se ha dicho, los protocolos a nivel de red más utilizados actualmente (IPX, IP y *AppleTalk*), así como los protocolos de nivel de enlace más conocidos : *Ethernet, Token-Ring,* ARCNET, FFDI).

Esto posibilita la interconexión de máquinas clientes con el servidor, independientemente del protocolo que las máquinas clientes utilicen para su conexión (siempre y cuando dicho protocolo sea soportado por el servidor). Para conseguir esto únicamente es necesario cargar todos los protocolos en la máquina servidor; de esta forma, cuando una máquina cliente realice una petición al servidor a través de un determinado protocolo, el servidor podrá reconocerlo y servir dicha petición.

El lector no debe pensar que basta con que el protocolo de mayor nivel sea igual para que la comunicación entre máquinas se produzca; para que un servidor se comunique con las máquinas clientes y con otros servidores debe soportar los mismos protocolos a varios niveles. Los niveles de protocolo importantes para los servidores son :

- Físico.
- Enlace de datos, con sus dos subniveles : método de acceso al medio y control de enlace.
- Red.

Para que dos sistemas se comuniquen directamente, deben utilizar el mismo protocolo para cada uno de los niveles citados. Si utilizan protocolos diferentes por debajo del nivel de red, será necesaria la presencia de un dispositivo (*router*) que solucione estas diferencias, permitiendo de este modo la comunicación.

El nivel más bajo, el físico, incluye características de hardware y señalización eléctrica. Como, por lo general, todos los aspectos del nivel físico se implementan por hardware, y como Netware está basado en el hardware de uso frecuente compatible con PC, se dispone de una gran variedad de hardware que implementa el interfaz con el siguiente nivel (enlace).

El subnivel de método de acceso al medio del nivel de enlace de datos define la transmisión de datagramas entre sistemas que comparten un enlace físi-

co. Los más destacables son 802.3 (*Ethernet*), 802.5 (*Token-Ring*) y control de enlace de datos de alto nivel (HDLC). Normalmente la mayor parte del protocolo de enlace de datos se realiza en hardware incorporado en la tarjeta de red a través de un chip controlador VLSI, aunque para algunas funciones de dicho nivel se necesita la ayuda de software. Algunos protocolos de enlace de datos, como el 802.3, pueden ejecutarse sobre medios físicos diferentes, por ejemplo 10-BASE-T o 10BASE5. Sin embargo, los medios diferentes no pueden conectarse directamente, sino que se necesita la presencia de algún dispositivo como puede ser un conversor, un repetidor o un *router*.

El subnivel de control de enlace define una forma de ejecutar varios protocolos de nivel de red en un sólo enlace. Entre los ejemplos se incluye el 802.2.

- Para Netware IPX :
- Protocolo de notificación de servicios (SAP).
- Protocolo de propagación de NetBIOS.

El diseño Netware ODI hace posible que los protocolos de red se combinen libremente con los tipos de tramas (encapsulación), garantizando de esta forma la compatibilidad con todas las combinaciones protocolo de red/tipo de trama. En la tabla 6 se muestran las combinaciones recomendadas.

CONFIGURACIÓN DE PROTOCOLOS

Además de cargar en el fichero de arranque apropiado del servidor los protocolos de red que éste va a soportar, es necesario configurarlos.

Para comprobar el funcionamiento del protocolo IPX, Netware incorpora la utilidad de consola IPXCON

Este nivel se implementa enteramente en software, y ya se ha visto cómo Netware soporta los tipos mas utilizados.

El nivel de protocolo de red permite la comunicación de las aplicaciones en enlaces LAN diferentes, teniendo en cuenta, por tanto, las posibles diferencias en el tipo de trama, enlace de datos o especificaciones físicas del medio.

Cada protocolo de red especifica también un protocolo de encaminamiento que los *routers* utilizan para conservar un mapa de la interred. Las estaciones de trabajo y los servidores no tienen que ejecutar el protocolo de encaminamiento a menos que funcionen también como *routers*. La tabla 5 muestra la relación entre tipos de máquinas cliente y los protocolos asociados.

Existe una familia de protocolos asociados a varios niveles para cada protocolo de red. Muchos de estos protocolos no entran dentro de categorías bien definidas, pero, a pesar de todo, los servidores deben soportarlos. Los más comunes son:

Configurar un protocolo de red supone darle los parámetros apropiados para su correcto funcionamiento. Para configurar un protocolo de red basta con:

1. Cargar la utilidad INETCFG.
2. Seleccionar la opción CONFIGURACIÓN E INTERCONECTIVIDAD.
3. Seleccionar la opción PROTOCOLOS.
4. Seleccionar la opción CONFIGURACIÓN DEL PROTOCOLO.
5. Seleccionar el protocolo que se quiere configurar (IPX, IP, etc.).
6. Realizar la configuración apropiada a dicho protocolo.
7. Salir.

ASOCIACIÓN DE UN PROTOCOLO A UN INTERFAZ DE RED

Después de configurar un protocolo, es necesario asociarlo a un interfaz de red; para ello los pasos que se deben seguir son los siguientes :

1. Cargar la utilidad INETCFG.
2. Seleccionar la opción CONFIGURACIÓN E INTERCONECTIVIDAD.
3. Seleccionar la opción ASOCIACIONES.
4. Pulsar *Ins* y seleccionar el protocolo que se quiere asociar (IPX, IP, etc.).
5. Seleccionar la tarjeta con la que se quiere asociar dicho protocolo.
6. Rellenar los parámetros solicitados, como son dirección de red, dirección de nodo, tipo de trama, etc.
7. Salir.

COMPROBACIÓN DE LA CONFIGURACIÓN

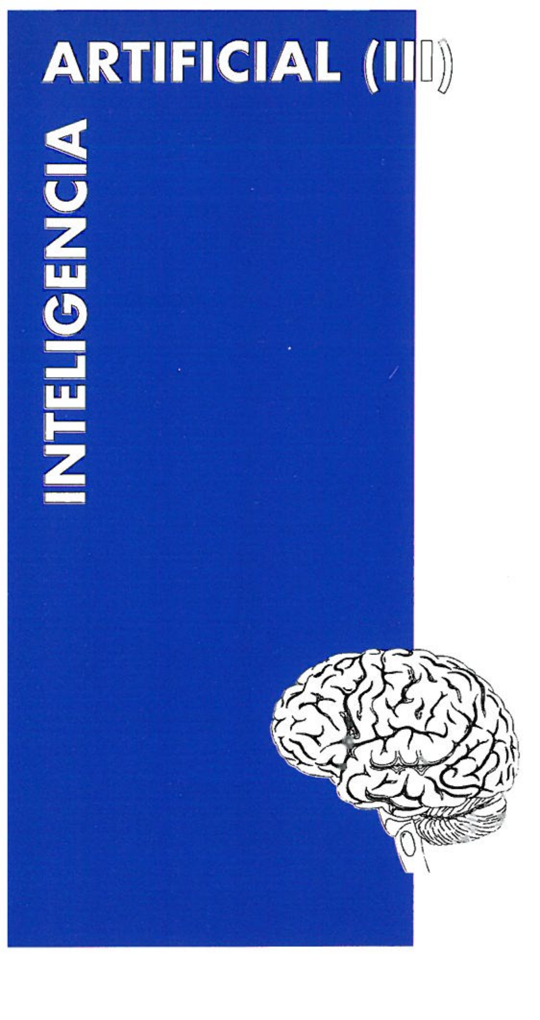
Una vez configurado el protocolo, se debe comprobar su correcto funcionamiento. Para ello, Netware dispone de algunas facilidades que permiten dicha comprobación; no obstante, debido a la mezcla de elementos software y hardware, esta labor no es sencilla.

Para comprobar el funcionamiento del protocolo IPX, Netware incorpora la utilidad de consola IPXCON, que muestra la dirección de la máquina, las estadísticas, el número de paquetes recibidos, enviados y reenviados, el número de circuitos locales activos, el número de redes conocidas, el número de servicios conocidos y la dirección del sistema que se está gestionando en este momento.

La utilidad para comprobar el correcto funcionamiento de la pila de protocolos TCP/IP se denomina PING. En este caso, la prueba puede lanzarse tanto desde una máquina cliente como desde el servidor.

PRÓXIMO NÚMERO

En el próximo número se explicará con mucho más detalle el mecanismo que Netware de Novell utiliza para gestionar varias pilas de protocolos, además de estudiar de forma más pormenorizada la configuración de éstos.



LOS SISTEMAS DE PRODUCCIÓN

Ramiro Carballo

Se presenta una nueva forma de programar: desaparece la secuencialidad, por ser demasiado rígida, debido a que las bifurcaciones sólo se efectúan en puntos y caminos, determinados a priori, en el código del programa. Sin embargo, con los sistemas de producción, los programas se organizan en base a los datos: el aprendizaje consiste en dotar al sistema de nuevos datos que dirigirán nuevas operaciones.

En un momento determinado de la historia de la Informática, los programadores se plantearon un problema: cómo conseguir que un programa, que funciona perfectamente con un conjunto de datos determinado (gracias al conocimiento que le hemos proporcionado), funcione también con otros datos con los que todavía no se ha encontrado, y para los que debe realizar unas operaciones que todavía no han sido implementadas en su código.

La solución reside en el aprendizaje, en cómo conseguir que el programa se adapte a los nuevos datos que le van llegando. Esto implica, directamente, la modificación del programa en tiempo de ejecución, para que pueda responder con nuevas acciones a nuevas situaciones.

Los programas tradicionales tienen la peculiar característica de la secuencialidad. El flujo del programa depende del valor de los datos que son testeados en un momento determinado del código, en unas bifurcaciones fijadas a priori en tiempo de diseño, y de difícil "autoactualización", para el propio programa, en tiempo de ejecución. La secuencialidad, a la que todo el mundo está acostumbrado, se ha convertido en un defecto (a pesar de que se trate de una estructura adecuada para ciertos tipos de computación).

Se llega a la conclusión de que la mejor manera de conseguir que un programa adquiera la posibilidad de hacer algo que no haya podido hacer antes, es que se pueda añadir código a sí mismo. Sin embargo, en los programas tradicionales nos encontramos con procedimientos, argumentos y variables, cuyo significado se ha de conocer para poder modificar el programa.

Por otra parte, también es necesario entender el propósito de cada línea de código. Se trataría de que el programa, que quiere aprender, esté en condiciones de modificar código (sin introducir errores), editar antiguos procedimientos, definir nuevos, etc. Sencillamente, difícil de automatizar.

Se podría idear un tipo de lenguaje que tuviese como virtud lo contrario que los tradicionales: en vez de ser secuencial, que la bifurcación fuese la norma. Que no sólo los datos, como tales, sean datos, sino también los sucesos de estado o la interacción con el usuario. Que el programa tenga un *mundo* definido, descrito por el conjunto de datos, y que lo examine en cada iteración, respondiendo con la acción asociada a cada estado del mundo (o contexto). El lenguaje, en sí, sería más primitivo de lo que estamos habituados, pero su sencillez también lo haría fácilmente modificable.

De esta manera, lo arriba expuesto, se denomina, generalmente, sistema de inferencia dirigido por patrones, siendo estos patrones los encargados de detectar los cambios que se produzcan en el contexto (los datos) y de lanzar la operación asociada. Esta operación, a su vez, puede modificar los datos, eliminando o añadiendo algunos de ellos, y consecuentemente, cambiando el modo de actuar del programa en sí, adaptándose a la nueva situación: aprendiendo.

Los sistemas de inferencia dirigidos por patrones se consideran, como hemos visto, un tipo de programa. Estos sistemas, se pueden clasificar en sistemas basados en redes y sistemas basados en reglas. Sin entrar en más detalles sobre cada grupo, estos últimos se subclasifican en sistemas de transformación (lógicos o gramaticales) y sistemas de producción (sobre los que versará este artículo).

Figura 1: Clasificación general de los Sistemas de Inferencia Dirigidos por Patrones. Los Sistemas de Producción constituyen un área concreta que será objeto de estudio

- Sistemas de Inferencia Dirigidos por Patrones
  - Sistemas Basados en Reglas
    - Sistemas de Transformación
      - Sistemas Lógicos
      - Sistemas Gramaticales.
    - Sistemas de Producción
      - Sistemas Dirigidos por el Antecedente
      - Sistemas Dirigidos por el Consecuente
  - Sistemas Basados en Redes

ESTRUCTURA GENERAL

Este tipo de sistemas podemos modelizarlos como una gran bolsa llena de datos. No todos los datos tienen por qué tener la misma estructura, es decir, existirán distintos tipos de datos (tantos como el usuario necesite definirse).

Por otro lado, disponemos de un conjunto de reglas. Cada regla, llamada de producción, consta de dos partes separadas por una flecha: la de la izquierda se denomina antecedente; la parte de la derecha es el consecuente.

El funcionamiento es el siguiente: para toda regla, se compara el antecedente de la regla con cada uno de los datos que tenemos en la bolsa. Si un dato coincide con la descripción de un antecedente de una regla (hecho que se denomina ajuste o equiparación), se dispara esa regla.

Una regla disparada produce la acción especificada en su consecuente. Esta acción puede consistir en la eliminación de un dato, la adición o la modificación del mismo. Dependiendo de la implementación o del entorno en que se ejecute, la acción podría ser una salida por pantalla, la ejecución de otra aplicación, etc.

Es evidente que en el proceso de ajuste pueden ser varias las reglas que cumplan la condición necesaria para dispararse. Es, entonces, necesaria la intervención o moderación, por parte de un árbitro, en el sistema.

Este elemento del sistema dará prioridad a unas reglas sobre otras, determinando, en cada momento, las reglas o conjunto de reglas que se dispararán entre todas las posibles.

Los tres elementos de los que consta un sistema de producción son:

a) Lo que denominamos base de hechos, base de datos, contexto, mundo o memoria de trabajo, que identifica el conjunto de datos del sistema.
b) La base de reglas de producción, que con sus acciones modifica la base de hechos.
c) El motor de inferencias, también llamado estrategia de control, intérprete de reglas, o máquina de deducción, que representa al árbitro que tiene criterios propios para decidir qué reglas se disparan de entre las posibles.

También se puede dar el caso en el que existan reglas que, en la acción especificada en su consecuente, generen otras reglas, ampliando el rango de operaciones que puede realizar el sistema.

Veamos cada uno por separado.

LA BASE DE HECHOS

Si nos proponemos desarrollar un sistema de producción para solucionar un problema concreto, debemos poner especial cuidado en el diseño de la base de hechos, para que el sistema pueda albergar la información adecuada para realizar esa tarea en particular.

Se especificará, en el conjunto *BHo*, los datos iniciales de los que se partirá, es decir, la base de hechos inicial. En la base de hechos final (*BHf*) se determinarán aquellos elementos que constituyen la meta de nuestro problema.

Debe quedar claro que se puede implementar de mil formas, y que para cada problema se debe escoger la más adecuada.

Sirva de ejemplo el desarrollo de un sistema experto que pudiese predecir, en la etapa de planificación de un proyecto informático para el que va a ser contratada una empresa de software, el tiempo que va a tardar esa empresa en llevarlo a cabo, especificando a priori el número de líneas de código de toda la aplicación, el número de horas de programador necesarias, e incluso el número de errores contenidos en la versión final. El sistema experto necesitaría manejar un gran bagaje de datos que reflejasen la experiencia de esa empresa en proyectos anteriores similares: características de cada proyecto, programadores, capacidades de los programadores, horas por programador, errores, momento en el que se introdujeron los errores en el proyecto, momento en el que se detectaron, etc.

La mejor implementación de la base de hechos de este sistema de producción sería una base de datos relacional, que pudiese gestionar rápidamente toda la información, y mantener las relaciones entre distintas tablas como la de proyectos, la de programadores o la de tipos de errores, por ejemplo. Debería albergar toda la experiencia de la empresa en proyectos anteriores.

Por otro lado, si se desease desarrollar un sistema de producción que jugase a las "tres en raya", nuestro elemento básico, en la base de hechos, sería un simple *array* de enteros de 3x3, que representase fielmente el estado del tablero en cada momento. Si las fichas de un jugador se detallasen como un 1 en el array, las del otro como un 2 y las casillas vacías como un 0, la base de hechos inicial vendría dada por el array con todos sus elementos inicializados a 0.

La base de hechos final especificaría aquellas configuraciones del tablero que determinan el final de la partida: tablero lleno o tres fichas de un jugador en raya.

En este caso no conviene detallar, en la BHf, todas las posibles configuraciones de tablero que acaban la partida, ya que llevaría bastante tiempo. La solución adecuada pasaría por el uso de una función que detectase las 3 en raya, o que detectase el tablero lleno, o, incluso, ayudar con un décimo elemento en el array que indicase el número de casillas llenas en el tablero.

El mismo problema de las tres en raya se puede afrontar con otra base de hechos más sencilla, que conste de una terna por cada casilla de tablero. Los dos primeros elementos de la terna pueden ser las coordenadas en el tablero, y el tercero, el contenido. Por convenio, las casillas vacías podrían especificarse (con un 0, por ejemplo, en el contenido), o simplemente omitirse en la base de hechos. De esta manera, la memoria de trabajo constará, como máximo, de 9 vectores de tres elementos cada uno.

Las bases de hechos formadas por elementos más pequeños, suelen ser más flexibles y resulta más fácil diseñar las reglas para manejar esos datos.

REGLAS DE PRODUCCIÓN

Si en la base de hechos se encontraban los datos del problema, en la base de reglas se almacena el conocimiento sobre la solución del mismo.

Las partes, que arriba denominamos como antecedente y consecuente, se corresponden fielmente con la condición y la acción de una estructura *if-then* de un lenguaje de programación estructurada.

Si se simplifica el concepto de regla de producción, la base de reglas de un sistema se podría implementar como una batería de estructuras *if-then-elsif...* Con un sistema así, se dispararía sólo la primera regla que cumpliese la condición.

Conviene aceptar un concepto más amplio: que la parte izquierda no sea una sola condición, sino una lista, al igual que en la derecha, una ristra de acciones.

Una regla, también llamada producción, verifica que se cumple una lista de condiciones, y, si es así, lanza o ejecuta una lista de acciones.

Figura 2: Estructura de un Sistema de Producción

BASE DE HECHOS	⟺ MOTOR DE INFERENCIAS ⟺	BASE DE REGLAS

También se debe contemplar en el concepto de regla, que se pueden conseguir razonamientos inexactos (ni sí, ni no) basados en la lógica difusa (*fuzzy*) en un sistema de producción, asociando a cada regla un factor de incertidumbre. Esto se puede utilizar en un sistema experto de medicina, que produzca diagnósticos alternativos con distinta solidez o certeza.

Todas las reglas deben cumplir la restricción de ser independientes entre sí, evitando hacer referencia unas a otras. Si una regla debe hacerle saber algo a otra, lo hará a través de la base de hechos, que es el único punto de unión entre las reglas de producción del sistema.

Esta independencia permite que el conocimiento del programa se distribuya por un conjunto "desordenado" de reglas (inconexas aparentemente). Esto favorece el mantenimiento del sistema, con tan sólo añadir, borrar o modificar reglas, ya que se da por supuesto que no existirán efectos colaterales entre ellas. Véase el ejemplo del árbol de decisión expresado en la figura 4 y compárese con la revisión de un algoritmo que realice una función similar: para actualizar, no es necesario un conocimiento profundo del sistema.

REPRESENTACIÓN DE LAS REGLAS

Sobre la forma de representar reglas, diremos que la lógica formal nos ofrece la más adecuada, sobre todo porque así pueden ser tratadas por un motor de inferencias general, como el del lenguaje *Prolog* (aunque implique restricciones en los tipos de fórmulas a utilizar).

Por nuestra parte, intentaremos simplificar las representaciones en favor de aquellos lectores menos familiarizados con la lógica formal.

Además, es suficiente distinguir las partes de una regla (condiciones y acciones) para poder implementarla en C, *Lisp* o en la herramienta *Prodigy*, adecuando la representación de las expresiones necesarias al lenguaje o herramienta concreto que se vaya a utilizar, aprovechando las facilidades que nos ofrezca.

Analicemos la representación de las reglas de producción de las "tres en raya"; en particular la variante que utiliza, como elemento básico de la base de hechos, las ternas. Estos elementos representaban las coordenadas de la casilla en el tablero, en el primer y segundo componentes y el contenido de la misma en el tercero. El contenido se representa con los valores *X* y *O*, para las fichas contrincantes, y *V* para las casillas vacías.

En este ejemplo, sólo se modeliza el movimiento de las fichas; no se contempla la alternancia entre jugadores, ni se busca la victoria.

Veamos primero una regla sencilla:
(2, 2, V) -> (2, 2, O) -(2, 2, V).

Esta regla, que en realidad representa la heurística que contempla el caso en el que la casilla central esté vacía, sólo se disparará si, en la base de hechos, se encuentra una terna que especifique las coordenadas (2, 2) y el contenido vacío.

En la acción, a la derecha de la regla, se observa que existen dos ternas, la primera sin signo (o con signo positivo, si así se quiere considerar) y la segunda con un menos delante. Un signo negativo delante de un elemento de la base de hechos, define la acción de eliminar ese elemento. Un signo positivo, o la ausencia de signo, determina la acción de añadir el elemento a la memoria de trabajo.

Es decir, si existe el elemento que describe la casilla central vacía, lo elimino y añado otro que indica que una ficha ocupa el lugar central.

No es éste el único convenio seguido para representar los elementos añadidos o eliminados de la base de hechos o memoria de trabajo. Otra



representación habitual sería la siguiente:
(2, 2, V) -> (2, 2, O),
donde los elementos considerados en el antecedente se eliminan siempre, y los del consecuente son añadidos. Por lo tanto, si se desea que un elemento del antecedente no sea modificado por la regla, se deberá reflejar también en la acción, para que vuelva a ser añadido. Se utilizará un convenio u otro, según sea más cómodo para cada problema concreto, pero siempre aclarando qué interpretación de la regla se debe seguir.

Otra regla:
($i, $j, V) -> ($i, $j, O) -($i, $j, V).

El carácter *$* precede siempre un nombre de variable. Igualmente se podía haber escrito *x,y* en lugar de *$i,$j.* El antecedente dispara la regla si existe en el tablero cualquier casilla vacía. Esto se consigue especificando una variable para la primera coordenada y otra para la segunda. Sólo el valor del contenido (*V = vacío*) está fijado en la condición de esta regla.

El funcionamiento del sistema de producción es el siguiente: en la etapa de equiparación, cuando se están buscando datos en la base de hechos, que se ajusten a los antecedentes de las reglas, se realiza la búsqueda sólo por los componentes constantes de la condición. Las variables, $i y $j, no tienen ningún valor definido antes de que se produzca el primer ajuste. Sin embargo, cuando se ha escogido un dato que cumpla la condición, se produce la instanciación de las variables, es decir, se asigna a las componentes variables de la condición los valores que tenía el dato.

Si en otro momento, dentro de la misma regla, se vuelve a referenciar una variable ya instanciada, aunque sea también en el antecedente, la variable tendrá ya un valor fijo (el que se le asignó anteriormente) y actuará como un componente constante del dato de cara a posibles equiparaciones.

Si esa variable, ya instanciada, aparece en otra regla, debemos considerarla una variable distinta. La variable no guarda el valor con el que ha sido instaciada, solamente durante la ejecución de la regla en la que se instanció. De lo contrario, no se cumpliría la condición de independencia entre reglas de producción.

En el consecuente, sólo se utilizan variables que hayan sido instanciadas en el antecedente de la misma regla, ya que en los consecuentes no se realiza equiparación y, por lo tanto, tampoco instanciación.

En el ejemplo, si existiese la terna *(3, 2, V)*, en la fase de equiparación el componente constante del antecedente (*V*) produce el ajuste de la condición con el dato. La variable *$i* toma el valor *3* y *$j* el valor *2*. La acción pretende añadir (*$i, $j, O*) y eliminar (*$i, $j, V*). Pero como la instanciación de las variables ya se ha producido (si no, no se hubiese disparado la regla), en realidad se está añadiendo *(3, 2, O)* y eliminando la casilla vacía *(3, 2, V)*.

Con esta regla, simplemente, se deposita una ficha en cualquier casilla que cumpla la condición de estar vacía. Una regla más elaborada puede intentar rellenar la última celda que falta para conseguir las "tres en raya", por ejemplo. Para ello, el antecedente debe exigir una relación entre varios elementos de la base de hechos. La regla podría ser:
($i, $j, O) ($i+1, $j-1, O) ($i+2, $j-2, V) -> ($i+2, $j-2, O) -($i+2, $j-2, V) (FIN)

En el antecedente se exigen condiciones para tres elementos. Implícitamente, se trata de la conjunción de las condiciones de existencia de los elementos en cuestión. Se puede convenir que la ausencia de operador lógico entre condiciones en un antecedente determina el *and* lógico de las mismas. Contrariamente, la disyunción debe ser explicitada, ya que no es común encontrar un *or* lógico en el antecedente de una regla, porque permitiría la división de la misma en dos más simples : *A V B -> C* equivale a *A -> C y B -> C.*

Por otro lado, en el consecuente, se advierte que se ha añadido a la memoria de trabajo un elemento *(FIN)*, que determina el final de la ejecución del sistema de producción, ya que se ha alcanzado una de las metas (tres fichas del jugador en raya). El uso de estos elementos de control es muy útil y debemos tenerlos en cuenta en la implementación de nuestro sistema. El lector no se debe complicar con el modo de representar un *(FIN)* en una base de hechos dominada por ternas que representen casillas. Hay múltiples soluciones a gusto del consumidor: la terna *(0, 0, 0)*, cualquier elemento sin sentido, un campo adicional en un registro que contiene el conjunto de ternas, etc.

Es de destacar que resulta más sencillo determinar el fin desde una regla que desde una función tradicional. Ello es debido a que son las reglas las que tienen el conocimiento para resolver el problema y este conocimiento es completo e independiente.

La otra característica importante de la regla utilizada en el ejemplo es el uso de la misma variable en varias condiciones del antecedente. El funcionamiento sería el siguiente: se busca una casilla que tenga la ficha "O", por ejemplo la *(1,3)*, con lo cual *$i=1* y *$j=3.* Para que se cumpla la segunda condición, debe existir el elemento *(1+1, 3-1, O)*, es decir *(2, 2, O)*, ya que la instanciación de las variables se realiza una sola vez en el antecedente. Si existe *(2, 2, O)*, todas las condiciones de la regla se cumplirán si además existe

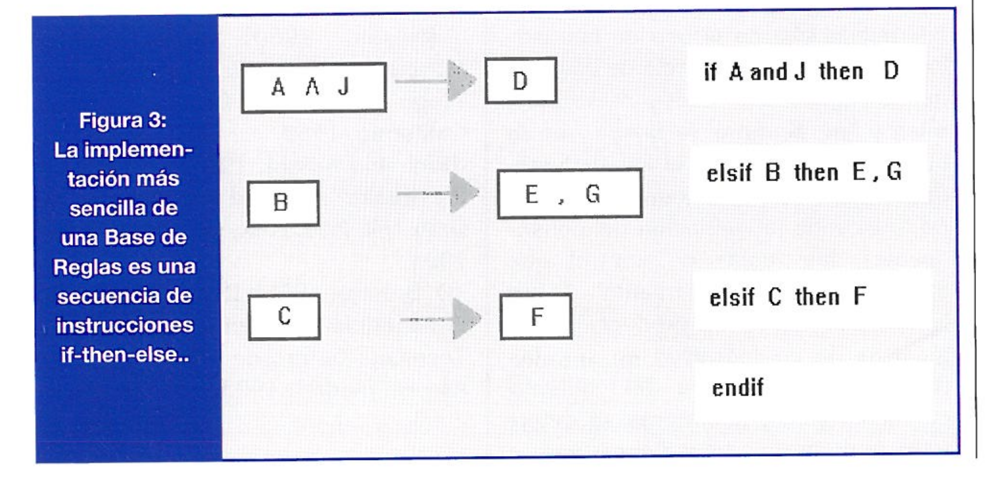


Figura 3: La implementación más sencilla de una Base de Reglas es una secuencia de instrucciones if-then-else..

(3, 1, V). Entonces se disparará la regla, ejecutando las acciones de añadir *(3, 1, O)* y *(FIN)*, y eliminar *(3, 1, V)*.

Por último, es interesante el estudio de la regla contraria, la que intenta arrebatar al contrincante la victoriosa consumación de las "tres en raya":

($i, $j, X) ($i, $p, X) ($i, $k, V) ($j <> $p) -> ($i, $k, O) -($i, $k, V)

Dos notas importantes en el antecedente: primero, que se pueden exigir condiciones cualesquiera entre elementos cualesquiera, mezclado con condiciones de existencia de datos; y segundo, la instanciación, en cadena, de las variables.

Suponiendo que exista *(1, 1, X)* en nuestra memoria de trabajo, resulta *$i=1*, *$j=1*. El siguiente ajuste viene determinado por *(1, $p, X)*. Suponiendo que existe *(1, 2, X)*, resulta *$p=2*. Análogamente, *$k* se instanciará con el valor adecuado existente en la base de hechos (si procede).

Al comprobar la última condición (*$j<>$p*) la instanciación (*1<>2*) determina el disparo de la regla. Se debe destacar que también existía la posibilidad de ajustar *($i, $p, X)* con *(1, 1, X)*, al igual que se hizo con *($i, $j, X)*, pero la última condición (*$j<>$p*) determinaba que se tratase de distintas casillas, para que se pudiese disparar la regla.

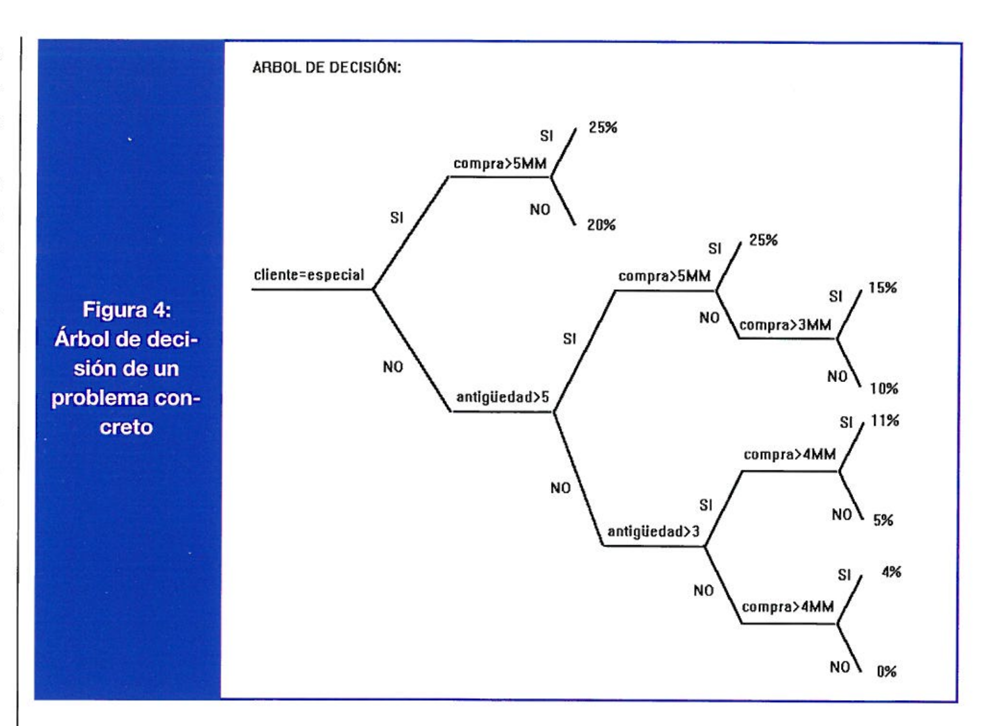


Figura 4: Árbol de decisión de un problema concreto

ESTRATEGIA DE CONTROL

La estrategia de control, o motor de inferencias, es el elemento del sistema de producción con capacidad para determinar qué regla se disparará de entre todas las que vean cumplida su condición. Se encarga, además, de activar reglas y encadenarlas en ciclos de funcionamiento. Se trata de un intérprete en su más puro concepto, como los de *Lisp* o *Basic*.

El procedimiento de control (que produce el secuenciamiento de producciones) de un motor de inferencias general es el siguiente:

a) La base de hechos toma el valor de los elementos especificados en la base de hechos inicial.
b) Si la base de hechos satisface la condición de terminación o se ha ejecutado alguna acción de parada, entonces FIN.
c) Se selecciona alguna regla R, en función de criterios propios, que pueda aplicarse al actual contenido de la base de hechos.
d) Se modifica la base de hechos con el resultado de la aplicación de la regla R.
e) Volver a b).

La estrategia de control resume, en definitiva, el alma del sistema de producción. Se comentará todo el abanico de posibles configuraciones, en próximos artículos, con el objetivo de ilustrar el funcionamiento interior de un motor de inferencias en general, dando la opción al lector de que implemente uno propio, adecuado al problema que desee tratar.

La complejidad del motor de inferencias abarca el uso de heurísticas para reducir el espacio de búsqueda de reglas, los modos de razonamiento (no siempre se busca que la base de hechos valide el antecedente, hay casos en los que ocurre al contrario, buscando la validación del consecuente), las reglas de control que albergan "metaconocimiento" sobre las propias reglas de producción, las técnicas de equiparación en sí y las técnicas de resolución del conjunto conflicto, donde finalmente se determina la regla que debe dispararse.

CONCLUSIÓN

En el próximo número se procederá a presentar una implementación de un sistema de producción que modelice el movimiento de las fichas de "las tres en raya", basado en lenguaje C. Comprobaremos las complicaciones que acarrea tener que considerar las actuaciones del motor de inferencia, ya que el compilador de C no proporciona ninguna de las funcionalidades de una estrategia de control integrada como ocurre en *Prodigy* o en el lenguaje *Prolog*. Se presentarán, también, las técnicas concretas usadas en la secuenciación de producciones y en la asignación de prioridades entre reglas.

Consultas al autor: carballo@lander.es

Referencias bibliográficas:

[Nilson, 1980]: N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co., Palo Alto, California, 1980

[Rich and Knight, 1991]: E. Rich and K. Knight. *Artificial Intelligence.* Mc Graw Hill, New York, segunda edición, 1991.

[D. Borrajo, 1993]: D. Borrajo y otros. Inteligencia Artificial. Métodos y Técnicas. Centro de Estudios Ramón Areces. Madrid. 1993.

Usuario Linux (I)

Teresa Madrid y César Sánchez

En UNIX y por extensión en Linux se sigue con la filosofía de tener elementos simples, cuyas combinaciones de diferentes maneras den como resultado soluciones a situaciones complejas. Otros sistemas de nacimiento más moderno han copiado algunas de las ideas de UNIX pero han dejado de lado esta idea básica, piedra de toque de su desarrollo, pagando como precio la pérdida de simpleza y elegancia en el resultado.

A lo largo de este capítulo nos basaremos en tres conceptos muy simples: multitarea, multiusuario y la información.

La multitarea es, como su nombre indica, varias tareas al mismo tiempo. Disponer de una arquitectura con varios procesadores sugiere de manera natural realizar más de una tarea al mismo tiempo. Aun con una sola unidad de ejecución, se verá la necesidad de ella y la justificación de incluirlo como uno de los elementos básicos del diseño.

La propiedad de ser multiusuario persigue que el sistema presente la infraestructura necesaria para que se pueda actuar con diferentes personalidades frente a él. Es decir, que ofrezca interfaces diferentes y ámbitos de actuación distintos para diferentes labores y para diferentes personas.

La pieza más importante de la computadora es la información que almacena. Se va a ir viendo a lo largo de esta entrega cómo la necesidad de tratar, gestionar, almacenar... está intrínsecamente ligada a la del diseño del Sistema Operativo.

LOS USUARIOS

Linux es un sistema multiusuario. Esto significa que varios usuarios comparten el sistema con su personalidad separada. Se tiene independencia en cuanto a las posibilidades de actuación y en cuanto a la información que el sistema maneja para ellos. Se debe destacar que, aunque la asignación de usuarios a personas surge de manera natural, puede ser œtil aprovechar esta separación para delimitar diferentes roles, actuaciones frente al sistema de información (sean una sola persona desempeñando varios roles o incluso varias personas físicas por cada uno). Anticipando conceptos, la administración del sistema (instalación de programas, configuración y demás tareas engorrosas que surgen en todo sistema informático) será asumida por una personalidad concreta, el superusuario. Ocultando así su complejidad y los problemas derivados de ella a los demás usuarios.

Cuando decimos que un sistema es multiusuario, éste debe cumplir una serie de requisitos. Uno de ellos es la separación de la información y con ello la privacidad de su trabajo. Otro de ellos es la compartición armoniosa de recursos del sistema. Cada usuario de un sistema Linux es reconocido por el mismo a través de una clave, comparte con el resto de usuarios las utilidades instaladas junto al sistema operativo y tiene reservado un espacio en disco que puede proteger y que puede ser limitado o no. Evidentemente el tamaño que ocupa un usuario siempre está limitado por el propio hardware. De lo que se habla es de anticiparse a la posibilidad de que un usuario acapare los recursos.

Los usuarios acceden utilizando un nombre de conexión o "*login name*", y son a través de él (re)conocidos por la



Una vez instalada una distribución de Linux, el siguiente paso es aprender a utilizar el sistema a nivel de usuario. Mientras se explican algunos conceptos sobre UNIX y sobre Sistemas Operativos en general se irán contando las órdenes más básicas y comunes.

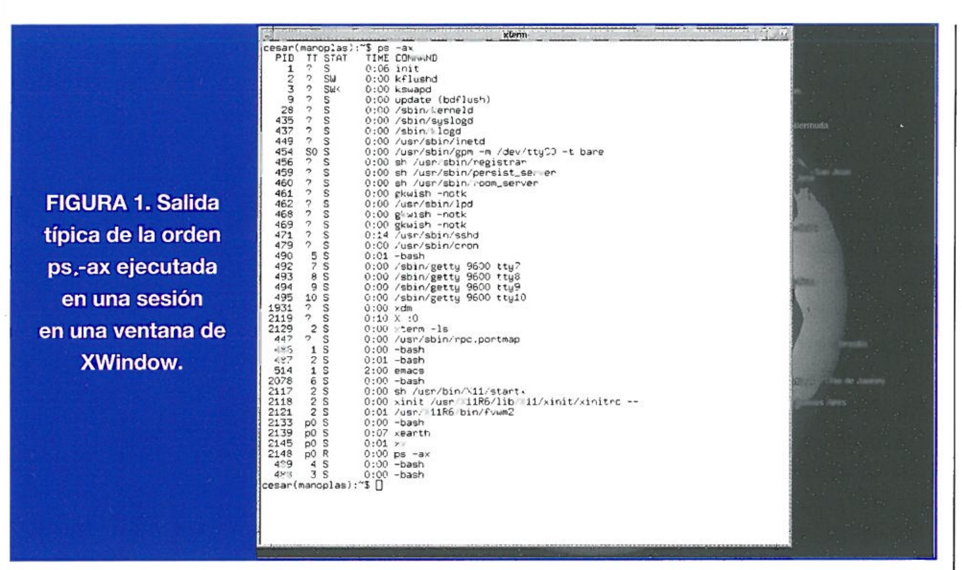


FIGURA 1. Salida típica de la orden ps.-ax ejecutada en una sesión en una ventana de XWindow.

máquina. Este nombre suele ser un nombre de no más de 8 letras formado con el nombre o apellido del usuario real. En cualquier caso debe tratarse de un identificador significativo del rol desempeñado. Al conjunto de recursos que el sistema asigna a un usuario se denomina cuenta (*login account*). En un sistema Linux habrá en general tantas cuentas como usuarios.

A lo largo del proceso de instalación de la distribución de Linux Debian se crearon dos cuentas. Por tanto al arrancar un Debian recién instalado existen dos usuarios autorizados a usar el sistema. Uno de ellos es el superusuario, con *login name root* que es el encargado de la administración. *root* es un usuario privilegiado, puede ejecutar órdenes que otros usuarios no pueden y acceder a todos los recursos. Esto significa, entre otras cosas, que *root* puede acceder a los ficheros de cualquier otro usuario. La labor del usuario *root* es de administración; la cuenta de *root* sólo debe ser usada con este propósito. De la administración se tratará en posteriores entregas.

El otro usuario creado es un usuario no privilegiado. Sólo tiene acceso a una parte de los recursos: aquellos que se le han asignado y los que comparte con el resto de usuarios. En este artículo supondremos que existen dos usuarios además de *root*. Los *logins* de estos usuarios, como ejemplo, asumiremos que son *tmadrid* y *cesar*.

AUTENTIFICACIÓN Y LOGIN

Para empezar a trabajar es preciso autentificarse. El proceso de autenticación consiste en acreditarse como un usuario válido. Por ello el sistema, al iniciarse, y tras culminar el proceso de arranque, muestra una invitación para introducir el *login*. Esta invitación suele tener un mensaje, indicando el nombre de la máquina y la versión del sistema operativo.

Al introducir el *login* de un usuario el sistema solicita una *password*. La *password* (palabra clave de acceso) es una clave secreta que a través de métodos de encriptamiento se comprueba como válida. Cada usuario debe tener y conservar como secreta esta clave, ya que es lo único que el sistema le pedirá para distinguirle de los demás. Tras introducir el *login* se procede a teclear la *password*, que, por motivos obvios de seguridad, no aparecerá en pantalla. Por ejemplo, un proceso de autenticación puede ser:

Bienvenido!
Este es un sistema GNU/Linux.

manoplas login: tmadrid
Password:

Como es natural cualquier usuario puede cambiar su propia *password*. Bastará con ejecutar la orden "*passwd*". Este programa solicitará la clave antigua y pedirá, tras la pertinente comprobación, una nueva clave. La antigua clave se pide como medida de seguridad, ya que si bien ya tuvo que introducirse al iniciar la sesión, siempre es posible que un intruso se acerque a una máquina en busca de una sesión abierta y olvidada, y podría cambiar la clave. La nueva *password* se pedirá por duplicado, para asegurar que no se hayan cometido errores al deletrearla. El programa *passwd* que se puede encontrar típicamente en un sistema GNU/Linux no dejará poner claves potencialmente inseguras, dados los actuales programas de ruptura de claves "*crackers*" y el algoritmo de encriptamiento usado. Se consideran claves potencialmente inseguras aquellas que contienen menos de ocho caracteres o que no incluyan nœmeros, símbolos o letras mayúsculas. El tema de la seguridad será tratado en profundidad en posteriores artículos de la serie.

SESIÓN Y SHELL

Tras autentificarse correctamente comienza una sesión. Se denomina sesión al periodo de tiempo desde que un usuario entra en el sistema (*log in*) hasta que deja de utilizarlo (*log out*). Durante el periodo que dura una sesión el usuario tendrá una cantidad de recursos asignados. En concreto dispondrá de un porcentaje de tiempo de ejecución de la CPU dependiendo de lo que quede disponible y la posibilidad de alterar y almacenar información.

Al comenzar la sesión, automáticamente se lanza un proceso que atenderá al usuario. Un proceso no es más que un programa en ejecución. De hecho la invitación a comenzar una sesión era también otro proceso. Los procesos tienen estado, y parte de ese estado es el dueño del proceso. El dueño del proceso encargado de atender tras la autenticación es el propio usuario que se autentificó. Poco a poco se irán conociendo más componentes que forman parte del estado de un proceso.

Puede notarse que el proceso de atención que ha sido automáticamente lanzado presenta un cursor parpadeante precedido por información sobre el nombre del usuario y la máquina; por ejemplo:

cesar(manoplas):~/lib/SoloProgramadores$ _

Esto es una invitación a introducir órdenes, más comúnmente conocido como *prompt*. A partir de este momen-

to se podrán introducir órdenes desde este *prompt*. La máquina procederá a ejecutarlas lanzando para ello nuevos procesos.

Este proceso que atiende se denomina *shell*. El equivalente a la *shell* en MsDOS es el conocido programa COMMAND.COM. La *shell* es un proceso fundamentalmente interactivo que recubre el sistema operativo y traduce las órdenes en llamadas al *kernel*. En otras palabras, es el proceso que muestra el ya mencionado *prompt* en la pantalla y espera que el usuario introduzca

Un sistema multiusuario permite actuar con diferentes personalidades frente a él

órdenes. Dichas órdenes pueden ser nombres de programas u órdenes propias de la *shell*.

Se pueden ver los procesos lanzados por el propio usuario ejecutando la orden *ps*:

cesar(manoplas):~$ ps
PID TT STAT TIME COMMAND
488 3 S 0:00 -bash
2086 3 R 0:00 ps
cesar(manoplas):~$ é

Efectivamente, se puede observar junto a otra información que existen dos procesos en ejecución pertenecientes a este usuario, *cesar*. Realmente hay muchos más procesos en ejecución. La mayoría de ellos pertenecen a *root* y fueron lanzados en el arranque. Son desde servidores de sistemas de ficheros de red a servidores de WWW o bases de datos, etcétera.

Se pueden ver todos los procesos en ejecución con las opciones -ax del programa *ps*. "*-a*" significa los procesos de todos los usuarios (*all*) y "*-x*" los procesos que no están ligados a un terminal (entrada de teclado y salida por monitor). Puede observarse en la figura 1 la salida una orden *ps -ax*.

Volviendo a los procesos del usuario *cesar*, obtenidos con *ps*, se observa que aparece el propio programa *ps* (como era de esperar, ya que es un proceso en sí mismo) y un proceso llamado *bash*. La *bash* es la *shell* anteriormente mencionada. Existen otras *shells* como *csh, tcsh, zsh*, pero coinciden en lo fundamental y la *bash* incorpora características muy interesantes.

En cualquier momento se puede dar por finalizada una sesión. Para ello hay que introducir la orden *exit* o la orden *logout* como respuesta al *prompt*.

MULTITAREA. CONSOLAS VIRTUALES

Se ha tenido ya oportunidad de observar las ventajas del diseño simple incorporando multitarea desde las bases del Sistema Operativo. Cuando se observó la orden "*ps -ax*" se pudo comprobar que había muchas otras tareas independientes siendo realizadas. Ya que eran tareas independientes es natural considerarlas como entes independientes dentro del sistema, y por ello eran ejecutadas como procesos separados.

El usuario también puede aprovechar el disponer de multitarea desde una propia sesión, lanzando un proceso e indicando a la *shell* que no deje de atenderle por el mero hecho de estar ejecutándolo en respuesta a una orden. El comportamiento normal de la *shell* es esperar a que éste termine. Puede comprobarse la diferencia entre:

cesar(manoplas):~$ sleep 10
cesar(manoplas):~$

que espera engorrosamente durante 10 segundos, y:

cesar(manoplas):~$ sleep 10 &
[1] 2218
cesar(manoplas):~$

que no lo hace. Es efectivamente el carácter "&" (*ampersand*) el que le indicó a la *shell* que no esperase. Se dice que la ejecución del proceso de esta manera se está realizando en "*background*" o entre bastidores.

Para aprovechar la multitarea, Linux dispone de una característica especial incluso dentro de los sistemas UNIX. Se trata de que dispone de consolas virtuales. Se denomina consola al teclado y monitor conectados directamente al ordenador. Se puede utilizar una máquina remotamente desde otro Linux u otros sistemas en red, o desde terminales conectados al puerto serie. Linux ofrece además la posibilidad de simular la existencia de múltiples consolas (teclado+monitor) utilizando la única consola de la máquina. En realidad lo que se hace es multiplexar la existente, y permitir la posibilidad de conmutar entre consolas virtuales. En un momento determinado sólo se dispone de una de ellas en la pantalla. Se puede cambiar de una consola virtual a otra pulsando la combinación de teclas Alt+F1 (pasaría a la consola 1) o Alt+F2 (pasaría a la 2), etc. Por defecto tras una instalación Debian se disponen de 6 consolas virtuales(Alt+F1 hasta Alt+F6) en cada una de las cuales se puede abrir una sesión diferente.

Como veremos posteriormente, el administrador puede configurar el sistema para disponer más consolas virtuales.

EL SISTEMA DE FICHEROS

Se han visto ligeras pinceladas acerca de Linux como sistema multitarea y multiusuario. Veamos ahora algunos aspectos relacionados puramente con el tratamiento de la información.

Linux al igual que muchos otros sistemas operativos dispone de un sistema de ficheros. Pero, ¿por qué se necesita un sistema de ficheros? El sistema de ficheros es el resultado de la necesidad de tratar información. La parte importante de una computadora es la información que se puede manejar con ella. Se debe disponer de un mecanismo de abstracción que independice del hardware subyacente y que permita operar con información compleja de la forma más sencilla posible. Este mecanismo es el sistema de ficheros.

Linux utiliza comúnmente el sistema de ficheros *extended 2 filesystem (e2fs)*, aunque permite acceder a otros sistemas de ficheros como los de *Windows95, MSDOS, OS-2, Apple* y muchos otros.

Un fichero es la agrupación de información, abstrayendo el medio físico en que está almacenada, bajo un nombre simbólico. Podemos abstraer informa-

ción de texto en un fichero de texto, o de un programa en un fichero ejecutable, o de descripción del hardware en ficheros de dispositivos. Los nombres de los ficheros en el *e2fs* pueden contener cualquier carácter, exceptuando el carácter í/í, y están limitados a 256 caracteres de longitud, no a los paupérrimos 8+3.

Nuevamente nos encontramos con una abstracción simple pero potente. En UNIX, el fichero tiene mucha descriptividad, no sólo representa almacenamiento en disco de bytes consecutivos. Los directorios son ficheros; el almacenamiento, por supuesto, también; la tabla de procesos, el estado de la máquina, los interfaces de red, la descripción del hardware... De esta manera, operaciones generales para el programador como abrir, cerrar, leer y escribir son extremadamente flexibles y potentes. No debe olvidarse que UNIX, y Linux especialmente, son sistemas diseñados por programadores para programadores.

Un directorio es un tipo especial de fichero, pero no por ello deja de serlo, que contiene índices a varios otros ficheros. De esto se deduce de forma limpia que los directorios se pueden anidar. Forman así una jerarquía que se denomina árbol de directorios. El árbol de directorios en UNIX es œnico, partiendo de un directorio que se denomina raíz y se representa por "/". Si se dispone de varias "piezas" de hardware de almacenamiento (discos), éstos formarán solidariamente parte de este único árbol de directorios. Se evita así la engorrosa tarea de tener que nombrarlos por separado teniendo además que incluir otra abstracción, otro concepto (como por ejemplo, la "unidad de disco" de MsDOS).

Para nombrar un fichero se debe indicar su localización dentro del árbol sin ambigüedad. Para ello, una primera manera de hacerlo es indicando toda la ruta dentro del árbol a partir del directorio raíz:

/home/cesar/documentos/carta1

Esta manera de nombrar los ficheros se denomina *path* absoluto. Por *path* se entiende la ruta de directorios para encontrar el fichero. Se puede nombrar también de manera relativa a un directorio concreto, que puede ser el directorio actual o un *ihome directoryí*. Se trata del *path* relativo.

El directorio actual forma parte del estado del proceso *bash*. Esto quiere decir que dicho proceso conoce en todo momento cuál es. El directorio actual se puede cambiar con el programa *cd (change directory)* y consultar con *pwd (print working directory)*.

cesar(manoplas):~$ cd /
cesar(manoplas):/$ cd /home
cesar(manoplas):/home$ pwd
/home
cesar(manoplas):/home$ _

Posteriormente se verán más ejemplos.

Para nombrar un fichero de manera relativa al directorio actual se escribe la ruta a partir de dicho directorio, sin estar precedido de la barra inicial. De este modo, si el directorio actual es */home/cesar* el fichero */home/cesar/documentos/carta1* del ejemplo anterior se nombraría también como:

documentos/carta1

El *ihome directoryí*, en adelante el *home*, es la parte de recursos de almacenamiento asociado a cada usuario. Cada usuario tiene, por tanto, un *home* propio. El *home* de cada usuario suele ser un directorio colgando de */home* y con el nombre del propio usuario. Así, por ejemplo, el *home* del usuario *cesar* será el directorio */home/cesar*. Al comenzar una sesión el directorio actual es el *home* del usuario.

Para nombrar un fichero con *path* relativo al *home* se emplea el carácter "~". De este modo *~cesar* es el directorio *home* del usuario *cesar*.

El proceso *bash* conoce en todo momento su dueño (es parte del estado), de este modo determina sin ambigüedad que el directorio nombrado con ~ es el de dicho usuario. Por tanto, para el usuario *cesar*, ~ es equivalente a *~cesar*. Resumiendo, existen varias formas de nombrar un mismo fichero. Para el usuario *cesar* y el directorio actual */home/cesar* son equivalentes:

/home/cesar/documentos/carta1
~cesar/documentos/carta1
~/documentos/carta1
documentos/carta1

PRIMEROS PASOS

La mejor manera de aprender a utilizar Linux es a través del "*hands on imperative*", es decir, practicando. Una ayuda importante son las páginas del manual, que aunque se les saca mayor rendimiento siendo usuario avanzado ofrecen una descripción "*on line*" de las órdenes, utilidades y librerías de programación.

Para acceder a las páginas de manual se utiliza la orden *man* seguida del término sobre el que se solicita ayuda. Por ejemplo:

cesar(manoplas):~$ man man

para la propia orden *man*, o

cesar(manoplas):~$ man cd

para *cd*.

Como ya se ha adelantado, el modo de moverse por los directorios del árbol es con *cd*. *cd* es un programa que acepta un argumento, pero es opcional. La notación usual para representar esto (incluso en las propias páginas de manual) es:

cd [directorio]

El directorio se puede nombrar en cualquiera de las formas vistas más arriba (absoluta o relativa).

cd sin argumentos retorna al *home*. Por tanto son equivalentes:

cesar(manoplas):~$ cd
cesar(manoplas):~$ cd ~

Todo directorio contiene al menos dos directorios que son el directorio

Para acceder al sistema se usa un login y una password

nombrado con un punto "." y el nombrado con dos puntos "..". El directorio "." es el propio directorio, por tanto son equivalentes los directorios

documentos/carta/. y
documentos/carta

De este modo la orden "*cd .*" es correcta, aunque el resultado es aparentemente nulo: cambiar al directorio actual. Más interesante es el directorio ".." o directorio padre. Es el directorio inmediatamente superior al actual. De este modo son equivalentes los directorios

Durante la sesión el usuario tendrá recursos asignados

/home/cesar/documentos/.. y
/home/cesar

El resultado de la orden "*cd ..*" es cambiar al directorio padre.

cesar(manoplas):~$ cd
cesar(manoplas):~$ cd documentos
cesar(manoplas):~/documentos$ pwd
/home/cesar/documentos
cesar(manoplas):~/documentos$ cd ..
cesar(manoplas):~$ pwd
/home/cesar

Para ver el contenido de un directorio se utiliza el programa *ls*. El programa *ls* puede tomar dos, uno o ningún argumento.
ls [opciones] [fichero]

ls sin argumentos lista el contenido del directorio actual. *ls* con un nombre de fichero muestra información sobre ese fichero. *ls* seguido del nombre de un directorio muestra el contenido de ese directorio.
cesar(manoplas):~$ pwd
/home/cesar
cesar(manoplas):~$ ls
documentos
cesar(manoplas):~$ ls documentos
carta1   factura

Se puede notar que el directorio í.í y el directorio í..í, que según se dijo existen en todo directorio, no aparecen listados. Esto es así porque su nombre comienza por un punto. Todos aquellos ficheros cuyo nombre comienza por un punto no aparecen listados al ejecutar *ls*, y por ello se denominan ocultos. Para listar los ficheros ocultos es preciso utilizar la opción *-a* del programa *ls*.

cesar(manoplas):~$ ls -a
.   .alias   .bash_profile .cshrc
..   .bash_history .bashrc

Se observa que en el *home* aparecen, además de los directorios í.í y í..í, algunos ficheros cuyo nombre comienza por punto y cuya utilidad se comprenderá más adelante. Los ficheros ocultos no tienen ninguna característica especial, salvo que *ls* no los lista por defecto, y cualquier usuario puede crear ficheros y directorios de este tipo.

Otra opción interesante de *ls* es *-l*. Con esta opción los ficheros se listan con información adicional, el dueño del fichero, su tamaño, la fecha de la última modificación e información acerca de sus permisos. Los ficheros, como los procesos, tienen dueño. De este modo se puede restringir la lectura, escritura y ejecución del fichero al dueño o al resto de los usuarios.

Para crear un directorio se utiliza *mkdir*, que requiere como argumento el nombre del directorio a crear. Para eliminar el directorio se emplea *rmdir*, que toma igualmente el nombre del directorio. El directorio ha de estar vacío para poder ser borrado.

mkdir [directorio]
rmdir [directorio]

Para borrar ficheros hay que utilizar *rm*. *rm* puede tomar tantos argumentos como ficheros se quieran borrar, además acepta opciones:

rm [opciones] [fich1] [fich2] ...

Por ejemplo:
cesar(manoplas):~$ cd ~/documentos
cesar(manoplas):~$ ls
carta1   factura
cesar(manoplas):~$ rm carta1 factura

rm no pregunta antes de borrar, aunque se puede hacer que pregunte utilizando la opción *-i*. Los ficheros borrados en UNIX son irrecuperables, por tanto conviene ser cuidadoso, sobre todo si el usuario es *root*, que puede borrar cualquier fichero.

Con la opción *-r rm* borrará los ficheros recursivamente, adentrándose en los directorios y borrando también éstos. En resumen, elimina el árbol de directorios por debajo del directorio pasado como argumento.

Se puede examinar el contenido de un fichero utilizando un paginador (*pager*). Al igual que los editores permiten modificar el texto de un fichero de texto los paginadores permiten sólo consultarlo. *more* es un paginador idéntico al de MsDOS. *less* es otro de ellos. *less* permite desplazarse por el contenido del fichero con las teclas del cursor, la barra espaciadora avanza una página completa. Pulsando "/" seguido de una cadena de caracteres y ENTER se busca dicha cadena. Con la tecla "n" se busca la siguiente ocurrencia después de haber realizado la primera búsqueda. Para salir de *less* se debe pulsar la tecla "q". Se puede obtener más ayuda mediante la tecla "?".

Se pueden realizar búsquedas de patrones desde la línea de comandos con *grep*. Aunque *grep* es muy completo (véase la página de manual), básicamente funciona segœn:
grep [patron] [fichero]

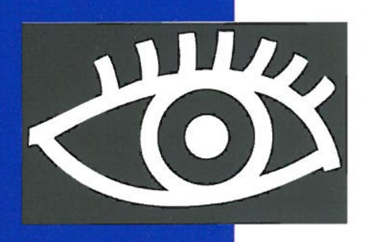
como en:
cesar(manoplas):~$ grep cesar /etc/passwd
cesar:QZcL9lqt9SSdw:1000:1000:Cesar ,,,,:/home/cesar:/bin/bash
cesar(manoplas):~$ _

que saca la línea correspondiente al usuario *cesar* en fichero de claves */etc/passwd*.

A lo largo de las próximas entregas se irá profundizando en cómo sacarle mayor rendimiento como usuarios sobre todo desde el punto de vista de los programadores así como aprender a administrarlo.

PROGRAMACIÓN DE CONTROLES OCX

Jorge R. Regidor



Desde los orígenes de la programación se ha intentado buscar el método óptimo para que el esfuerzo invertido en la solución de un problema pueda ser aprovechado posteriormente en otros problemas similares.

Los métodos han sido muchos y variados, como la descomposición en funciones, librerías de funciones o módulos, pero sin embargo el gran avance en este terreno se está realizando con la programación orientada a objetos.

Dentro de Windows la reutilización de código empezó con las librerías de enlace dinámico (DLLs), los controles a medida (*custom controls*), los cuales se fueron especializando hasta llegar a los componentes reutilizables por antonomasia como son los controles VBXs. Estos controles se integran entre otros con *Visual Basic* permitiendo crear desde este entorno aplicaciones basadas en componentes creados desde *Visual C++*. Quizá esta propiedad permitió el desarrollo tan grande de *Visual Basic*, ya que desde un lenguaje muy sencillo y rápido en tiempo de desarrollo se podía acceder a funciones complejas encapsuladas bajo estos controles VBX y codificadas en *Visual C++*.

Cuando Windows dio el paso a los 32 bits de la mano de *Windows NT* y *Windows 95*, los componentes VBX se transformaron en componentes OCX, aprovechando todas las capacidades ofrecidas por OLE y dando un paso más en la reutilización de código. Los controles OCX o controles OLE pueden ser utilizados desde cualquier aplicación contenedora OLE como son *Word, Excel, Access* y otras aplicaciones a desarrollar. Esto permite crear componentes OLE de las partes comunes a nuestros proyectos e integrarlos en ellas, así como adquirir componentes de otros desarrolladores de software; de hecho existen empresas dedicadas única y exclusivamente al desarrollo de componentes OCX reutilizables.

Un control OCX es un componente software reutilizable que soporta gran parte de la funcionalidad de la especificación OLE y que puede ser personalizado para adaptarse a casi cualquier tipo de aplicación.

ACERCA DE OLE

OLE comenzó como un método para la comunicación entre las aplicaciones Windows, y poco a poco se fue convirtiendo en toda una especificación de intercomunicación de objetos dentro de este sistema operativo. Hablar sobre OLE podría llevar artículos y artículos, por lo que aquí se va a describir el proceso para crear controles OCX, los cuales van a realizar unas determinadas tareas y van a ser controlados desde una aplicación con el método descrito posteriormente. Como se observará, el método para la creación de controles OLE ha sido radicalmente simplificado desde *Visual C++ 4.0*, lo que va a permitir crear sofisticados controles OCX sin necesitar profundos conocimientos de OLE.

COMPONENTES BÁSICOS DE UN CONTROL OLE

Un control OLE (OCX) puede ser utilizado desde cualquier tipo de aplicación contenedora OLE, como es *FoxPro 3.0, MS Word 6.0, Visual Basic 4.0* o el propio *Visual C++ 4.0*. Estas aplicaciones contenedoras se comunican con los OCX mediante las propiedades del control, los métodos y los eventos de notificación.

- Las propiedades del control son valores exportados por el OCX que pueden ser establecidos y/o consultados desde la aplicación contenedora.
- Los métodos son llamadas a funciones del control que realizan determinadas acciones, normalmente en función del estado de sus propiedades.
- Los eventos de notificación permiten a la aplicación contenedora ser informada de determinados sucesos, mediante el disparo de eventos, los cuales deben ser tratados por la aplicación contenedora.

Con todo esto es posible determinar que el modo natural de operar con los controles OLE es:

1. Fijar las propiedades deseadas sobre el control, así como determinar el estado del control mediante la revisión de dichas propiedades.
2. Realizar las operaciones deseadas sobre los métodos exportados por el control OLE. Obviamente sólo se pueden llamar a los métodos exportados por el control.
3. Atender a las notificaciones del control que van a informar de los sucesos notables dentro del mismo.

Estos tres mecanismos deben de ser perfectamente asimilados, ya que son el instrumento de comunicación entre los controles OCX y las aplicaciones contenedoras OLE.

CREANDO NUESTRO OCX CON APPWIZARD

Visual C++ 4.0 facilita de forma notoria la creación de controles OCX mediante la herramienta *AppWizard*, la cual incluye una opción específica para la creación de controles OLE. Esta opción permite crear la estructura básica sobre la cual se va a basar todo el trabajo de desarrollo. Además desde la herramienta *ClassWizard* se va a automatizar también en gran medida la creación de las variables y funciones miembro de la clase encargada de gestionar el control y que van a permitir almacenar tanto las propiedades como los métodos del control OLE.

Para comenzar un nuevo proyecto se selecciona del menú File la opción New, o bien se pulsa CTRL+N. En el diálogo New se selecciona la opción Proyect Workspace, lo que da lugar a iniciar AppWizard. Una vez dentro de AppWizard se debe seleccionar la opción OLE ControlWizard, opción que lanza el asistente para crear un control OCX.

El primer paso del asistente (ver figura A) permite indicar el número de controles que se quieren incluir bajo el mismo fichero OCX, lo que permite tener librerías de controles. Además se puede indicar si va a existir licencia de ejecución; en caso afirmativo, Visual C++ creará automáticamente un sistema de control de licencias de ejecución. Por otra parte se puede seleccionar si se quiere que el código generado por AppWizard sea con comentarios o no, así como la posibilidad de la creación de los ficheros de ayuda sobre el control OLE.

En el segundo paso (ver figura B) se pueden seleccionar otras diferentes opciones para cada uno de los controles OLE incluidos en el proyecto. Estas opciones son la posibilidad de que el control se active al visualizarse dentro de la aplicación contenedora y la capacidad para ser invisible en tiempo de ejecución, lo que no implica que sea invisible en tiempo de edición o diseño desde la aplicación contenedora. La opción insert dialog permite que el nombre del control sea incluido dentro del diálogo de inserción de objetos de la aplicación contenedora. About Box permite que el control contenga un diálogo de descripción del control y su autor. La selección de As a simple frame control crea un control que sirve como contenedor de otros controles, pero sirve como interfaz exterior.

En la lista Control subclass se puede seleccionar entre los diferentes controles estándar de Windows, lo que permite ampliar la funcionalidad básica de cualquiera de estos controles de Windows.

La ventana de edición de nombres se presenta al pulsar el botón Edit Names, y muestra y permite editar, como se observa en la figura B, los diferentes nombres utilizados por el control, desde el nombre del objeto OCX a registrar, el nombre de las clases C++, los ficheros fuente y de cabecera, etc

ESTRUCTURA DE CLASES Y ARCHIVOS

Una vez terminada la ejecución de AppWizard, se crean una serie de ficheros fuente, dentro de los cuales están las siguientes clases:

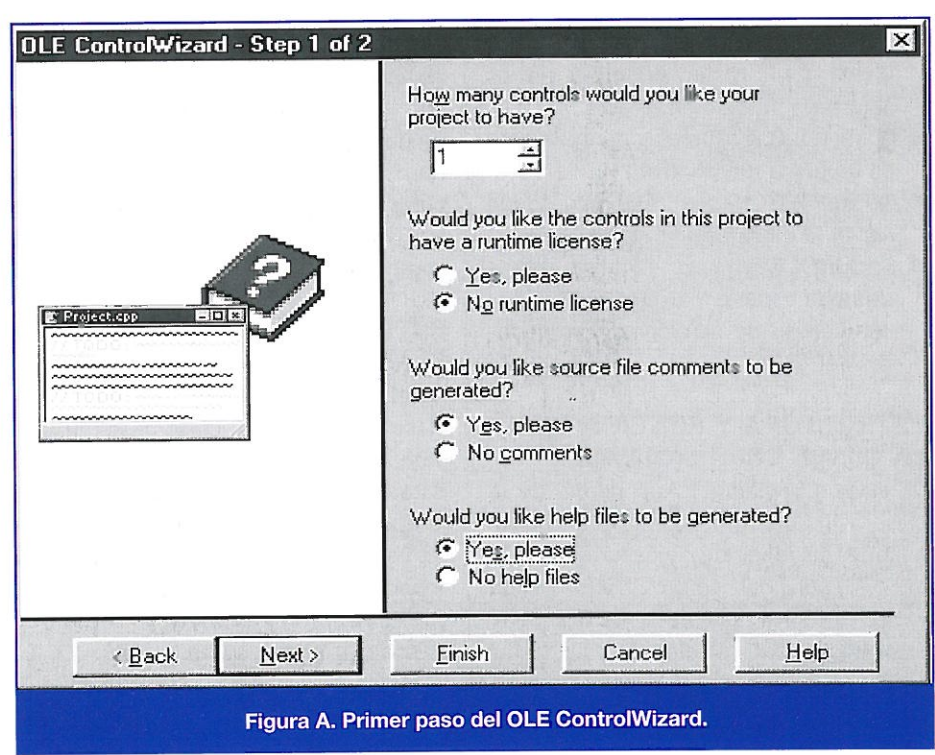


Figura A. Primer paso del OLE ControlWizard.

- CxxxApp: Clase encargada de controlar la carga y descarga de la DLL, ya que un control OCX no deja de ser una DLL especializada.
- CxxxCtrl: Esta clase derivada de la clase COleCtrl encapsula todo el proceso de comunicación y funcionalidad del control.
- CxxxPropPage: Esta clase permite controlar el acceso al diálogo de edición de propiedades del control.

Donde xxx suele indicar el nombre seleccionado para el control OLE, a no ser que se haya modificado dentro del diálogo de edición de nombres.

Los ficheros fuente significantes dentro del proyecto creados por AppWizard son (suponiendo que el control se llama SocketOLE):

- Readme.txt: Fichero donde se describen todos los demás ficheros.
- SocketOLE.cpp: Fichero donde está definida la clase CSocketApp que controla la carga/descarga de la DLL.
- SocketOLE.def: Fichero de definición del tipo de aplicación, en este caso una DLL.
- SocketOLE.odl: Este fichero contiene el código fuente en formato ODL (Object Description Language) que permite crear el fichero TBL que describe el control. Este fichero es modificado automáticamente por Visual C++, y en principio no es necesaria su edición directa.
- SocketOLE.rc: Fichero de definición de recursos del control.
- SocketOLECtl.cpp: Fichero donde reside la clase principal del control.
- SocketOLEPpg.cpp: Fichero donde reside la clase encargada del control del diálogo de edición de propiedades del OCX.

ESTRUCTURA Y FUNCIÓN DE LA CLASE CSocketOLEApp

La clase CSocketOLEApp deriva de la clase COleControlModule, la cual contiene todas las funciones necesarias para la inicialización del control. Sólo puede existir un objeto por cada OCX y debe estar declarada a nivel global. De todos modos todas las operaciones necesarias sobre esta clase están ya realizadas por AppWizard.

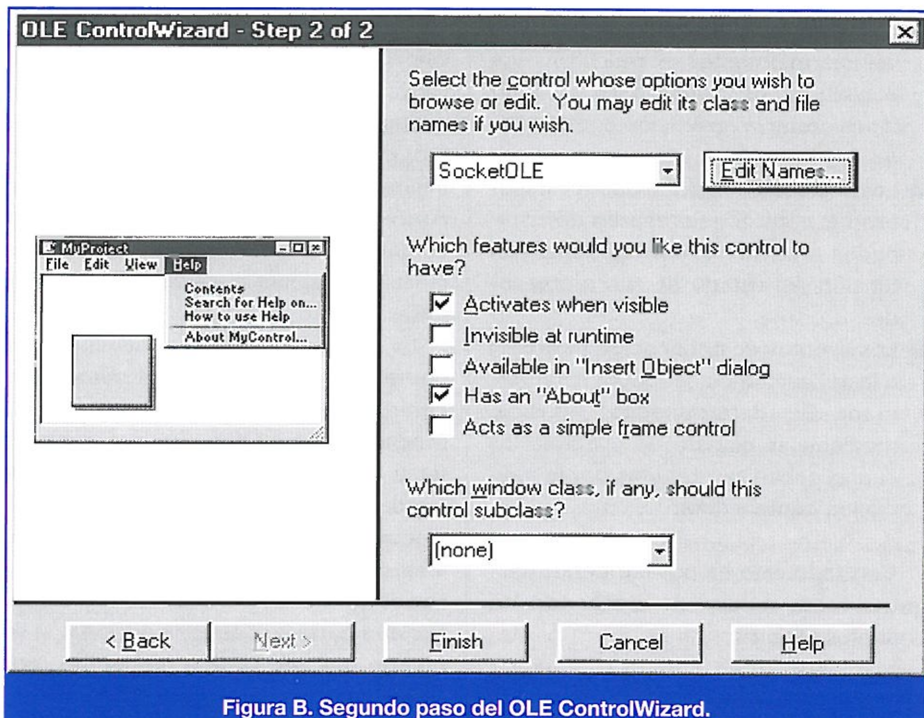


Figura B. Segundo paso del OLE ControlWizard.

La clase CSocketOLEApp deriva de la clase COleControlModule. En esta clase se sobrecargan dos funciones de la clase CWinApp (la clase "abuela" de CSocketOLEApp y "madre" de COleControlModule) que son InitInstance y ExitInstance. La función InitInstance pertenece a la clase CWinApp y es llamada cada vez que se inicia una nueva instancia de la tarea, es decir, cada vez que se carga el control. Dentro de esta función se pueden realizar operaciones de inicialización anteriores a la carga del control. La función ExitInstance es la función complementaria a la anterior dentro también de la clase CWinApp y permite realizar algún tipo de operaciones antes de finalizar completamente la ejecución del control.

ESTRUCTURA Y FUNCIÓN DE LA CLASE CSocketOLECtrl

La clase CSocketOLECtrl deriva de la clase COLEControl, clase base utilizada para la elaboración de controles OLE. Esta clase además deriva de CWnd, por lo que hereda toda la funcionalidad de las ventanas añadiendo la funcionalidad específica de los controles OLE, como son el disparo de eventos, el soporte de métodos y propiedades.

Mediante esta clase se consigue la comunicación entre las aplicaciones contenedoras y el control, realizando una conexión entre las propiedades y los métodos del control con los datos y las funciones miembro de la clase. Esto además está muy integrado, como se verá posteriormente con la herramienta ClassWizard, lo que simplifica mucho el trabajo.

La estructura creada por defecto con AppWizard crea una clase con las funciones miembro OnDraw, DoPropExchange, OnResetState, AboutBox. La función miembro OnDraw es llamada por Windows cuando el área de clientes necesita ser redibujada, por ejemplo, cuando una ventana, estando encima del control, ha sido movida a otro sitio. La función DoPropExchange sirve para implementar el interfaz persistente del control, mediante el cual se pueden almacenar en disco las propiedades que se quieran, por lo que no necesitan ser inicializadas cada vez que se carga el control. La siguiente función, OnResetState, es llamada cuando se le indica al control que restablezca su estado por defecto. Es en esta función donde debemos indicar los valores por defecto que deseemos para las variables miembro de la clase, o lo que es lo mismo, los valores por defecto de las propiedades del control. Por último, la función miembro AboutBox es llamada cuando se llama a la función miembro

AboutBox, que muestra un diálogo donde normalmente se muestran los créditos e información relevante del control. Además de las funciones miembro, se crean el constructor y el destructor. Dentro del destructor se inicializan los índices del control OLE, y se pueden incluir todas las inicializaciones de cada instancia al control que se creen. Un sitio idóneo para inicializar las variables miembro. En el destructor no existe ninguna acción por defecto, pero puede ser para liberar posible memoria reservada, etc.

ESTRUCTURA Y FUNCIÓN DE LA CLASE CSocketOLEPropPage

La clase CSocketOLEPropPage permite controlar el diálogo donde se pueden configurar todas las propiedades del control en tiempo de edición del mismo. Esta clase está asociada con el diálogo IDD_PROPPAGE_SOCKETOLE, donde se pueden añadir todos los botones, check-boxes, entradas de texto, etc, necesarias para la modificación de las propiedades del control.

AÑADIR NUEVAS PROPIEDADES

La manera más sencilla de añadir nuevas propiedades al control pasa, como se comentó con anterioridad, por utilizar la herramienta ClassWizard. Para ello se abre el diálogo de control de ClassWizard pulsando las teclas CTRL+W, o bien seleccionando la opción ClassWizard del menú View. Al mostrarse el diálogo se debe seleccionar el apartado OLE Automation, y para añadir una nueva propiedad pulsar el botón Add Property.

Una vez pulsado este botón, se muestra un diálogo (Ver figura D) que permite introducir todas las opciones de la propiedad. En la zona superior, campo External Name, se puede introducir el nombre de la propiedad, tal y como va a ser visto desde las aplicaciones contenedoras; en el ejemplo se ha introducido el nombre IPAddress. En el siguiente campo, Type, se puede indicar el tipo de la variable miembro que va a ser asociada con la propiedad, en este caso se va a seleccionar CString. El siguiente campo es Variable Name, donde se debe indicar el nombre de la variable miembro asociada a la propiedad. ClassWizard crea la variable automáticamente, seleccionando el tipo del campo anterior. En este caso se indica el nombre m_IPAddress. El siguiente campo permite indicar el nombre de una función llamada cada vez que se detecte un cambio en la propiedad. Esto permite realizar una serie de acciones en función del valor indicado para la propiedad. Por ejemplo en este caso se podría validar el formato de la dirección IP, de tal modo que si se observa que el formato no se corresponde con X.X.X.X, donde X es un número entre 0 y 255, se puede mostrar un mensaje de error indicando dicha circunstancia.

El siguiente apartado, Implementation, muestra el método de implementación de la propiedad dentro de la clase. Existen tres métodos de implementación que son Stock, Member Variable y Set/Get methods.

- Stock: Es un método sólo disponible para un grupo de propiedades más o menos comunes y que tienen implementado un método de atención ya por defecto.
- Member Variable: Es el método más común que asocia la propiedad directamente a la variable miembro indicada.
- Set/Get methods: Permite cambiar una propiedad mediante dos funciones miembro, una para establecer su valor (Set) y otra para consultar su valor (Get). Dentro de este método, se pueden indicar una lista de parámetros para ser incluidos en las llamadas.

Una vez seleccionado el método, en nuestro caso Member Variable, se pulsa OK, lo que implica la creación de la propiedad dentro de la opción OLE Automation.

Después de esto a la clase CSocketOLECtrl se añade los siguientes miembros:

CString m_IPAddress: Es la variable miembro creada.

afx_msg void OnIPAddressChanged(): Es la función miembro encargada de atender los cambios de la variable anterior.

Además se añade la entrada dispidIPAddress = 1L en el enumerado donde se definen todas las propiedades del control, e indica el número de la propiedad.

INCLUSIÓN DE NUEVOS MÉTODOS

El proceso para añadir los nuevos métodos es similar al descrito anteriormente con las propiedades, y comienza también abriendo

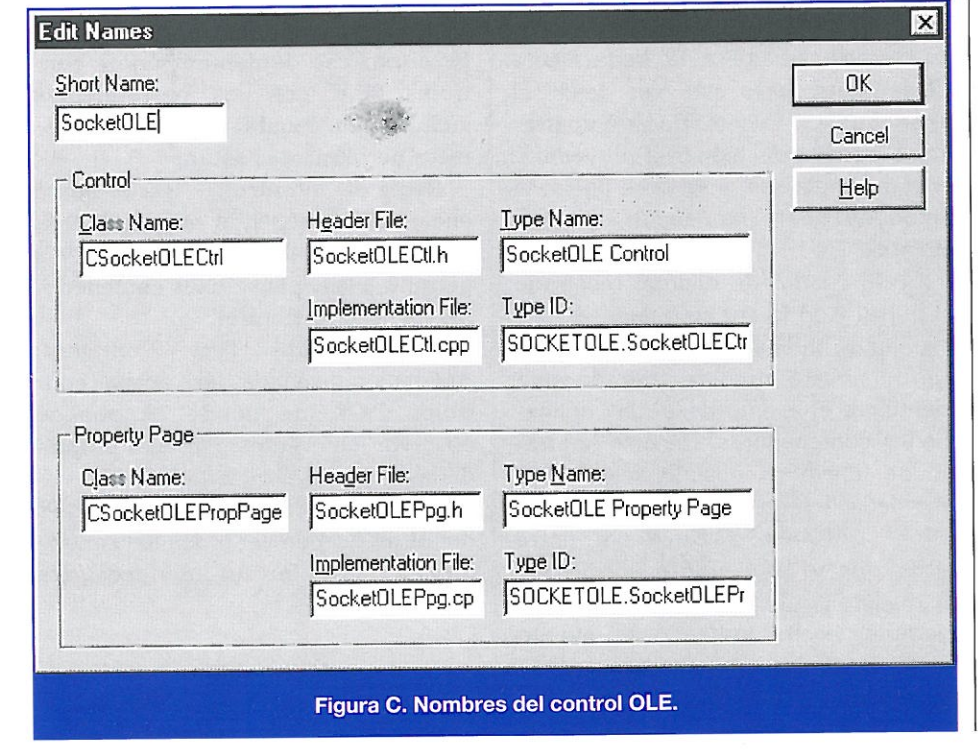


Figura C. Nombres del control OLE.

Add Property

External name:	IPAddress	OK
Type:	CString	Cancel
Variable name:	m_IPAddress	Help
Notification function:	OnIPAddressChanged	

Implementation: Stock / Member variable / Get/Set methods

Parameter list: Name / Type

Figura D. Diálogo de Add Property.

ClassWizard y seleccionando el apartado OLE Automation. Dentro de esta opción hay que pulsar el botón Add Method.

En el diálogo para añadir métodos, el primer campo es External name, donde se debe indicar el nombre del método visto desde las aplicaciones contenedoras; en este caso hemos indicado Open. El segundo campo, Internal name, permite introducir el nombre de la función miembro de la clase. Además se puede indicar el tipo de retorno de la función, para informar sobre el resultado de la función. En el siguiente apartado, Implementation, se puede seleccionar elegir entre Stock y Custom, aunque la primera opción, al igual que con las propiedades, sólo está disponible para un conjunto predefinido de métodos. Estos pueden ser seleccionados en el primer campo, y DoClick y Refersh.

En la parte inferior del diálogo se nos permite introducir los diferentes parámetros de la función miembro, indicando el nombre y el tipo de los parámetros.

Por último, y fruto de pulsar OK se crea la siguiente función miembro:

```
BOOL CSocketOLECtrl::OpenSocket()
{
 // TODO: Add your dispatch handler
code here
 return TRUE;
}
```

como se puede ver, esta función tiene exactamente el formato que hemos indicado con anterioridad en el diálogo de ClassWizard, que ha permitido crear de modo sencillo este método.

INCLUSIÓN DE NUEVOS METODOS

Para la creación de eventos de notificación hacia la aplicación contenedora también se utiliza la herramienta ClassWizard, pero esta vez desde el apartado OLE Events. En este apartado se puede seleccionar el proyecto y la clase implicada, y se debe pulsar el botón Add Event para añadir un nuevo evento.

En el cuadro de diálogo mostrado al pulsar Add Event se puede, en primer lugar, indicar el nombre exterior para el evento, nombre utilizado para identificar el evento desde las aplicaciones contenedoras. En este apartado de nuevo es posible seleccionar uno de los eventos predefinidos como Click, DblClick o KeyPress entre otros, que se asociarán a un método de disparo por defecto (stock). Para el ejemplo se ha indicado el evento OnReceive. Posteriormente, se debe indicar el nombre de la función a la cual se va a llamar para lanzar el evento. En la parte posterior, se puede indicar los parámetros incluidos al lanzar el evento.

Una vez pulsado OK, se crean las siguientes líneas dentro de la clase CSocketOLECtrl:

```
// Event maps
//{{AFX_EVENT(CSocketOLECtrl)
void FireOnReceive()

{FireEvent(eventidOnReceive,EVENT_
PARAM(VTS_NONE));}
//}}AFX_EVENT
```

esto permite que cada vez que se quiera lanzar un evento se tenga sencillamente que llamar a la función FireOnreceive.

CONCLUSIÓN

En este artículo se ha pretendido realizar un seguimiento del proceso básico para la creación de un control OCX, sin entrar de lleno en tratar la especificación OLE, tema del cual existen libros específicos de importante complejidad. Frente a esto se ha pretendido ser prácticos y describir cómo se crea el control, proceso bastante sencillo y que permite comenzar a crear componentes reutilizables por las demás aplicaciones Windows.

EN EL SIGUIENTE ARTÍCULO

En el siguiente artículo se va a describir el proceso de depuración de controles OLE con las herramientas incluidas en Visual C++ 4.0. Dentro de estas herramientas están:

Register control: Herramienta encargada de registrar el control dentro del registro de Windows y que va a permitir a las aplicaciones contenedoras localizar el control.

OLE Control Test Container: Aplicación diseñada para probar controles OCX, de donde se pueden observar los eventos, cambiar propiedades y llamar a m_todos.

OLE Object View: Aplicación donde se pueden ver todos los controles OLE, así como sus interfaces para las aplicaciones.

MECANISMOS DE CONTROL DE FLUJO

Felipe Bertran y César Sánchez



En el capítulo de este mes se verán los sistemas de control de flujo de que dispone Ada. Dichos sistemas son los tradicionales bucles y sentencias condicionales que están presentes en otros lenguajes de programación. Sin embargo, se comprobará que la sintaxis de Ada los hace particularmente sencillos de entender.

Por flujo de ejecución se entiende el orden en el que se van ejecutando las sentencias de código. Las tres estructuras básicas de las que se dispone para este fin son: la sentencia *if*, la sentencia *case* y la sentencia *loop*. Existe otra más, la sentencia *goto*, pero su uso no se recomienda por motivos que se harán evidentes cuando se aborde su estudio. *If* y *case* sirven para ejecutar determinados bloques de código cuando se haga cierta una determinada condición. Por su parte, *loop* sirve para ejecutar un mismo bloque de código de forma repetida.

Es un buen momento para mencionar que Ada también dispone de artificios para ejecutar varios bloques de programa de forma concurrente. Ello significa que en nuestro programa se pueden estar ejecutando varias tareas al mismo tiempo. En otros lenguajes, esto se puede conseguir mediante librerías especiales que dependen de servicios proporcionados por el sistema operativo y que por lo tanto no están disponibles en muchas arquitecturas. En el caso de Ada, la concurrencia está integrada en el propio lenguaje, lo cual simplifica su tratamiento sobremanera. Como el ordenador sólo dispone de un microprocesador, en realidad, en cada instante de tiempo sólo se puede ejecutar una única instrucción. La sensación de ejecución simultánea de varias tareas se consigue haciendo que el microprocesador vaya ejecutando trozos de las distintas tareas de forma alternada. Ello se hace a tal velocidad que parece que todo se ejecuta a la vez. En este capítulo todavía no se verá la concurrencia, aunque esté incluida dentro de lo que se considera control de flujo. Este mes únicamente se verán las estructuras de control secuencial.

SENTENCIA IF

Cuando se quiere que una determinada secuencia de instrucciones se ejecute sólo en el caso de que una condición sea cierta, se utiliza una sentencia *if*. Una sentencia de este tipo empieza con la palabra reservada *if* seguida de una expresión *booleana*, seguida de la palabra reservada *then*. A continuación se especifica la secuencia de instrucciones que se quiere que se ejecute en caso de que la expresión sea cierta, y se termina todo ello con las palabras *end if* y un punto y coma. Una expresión *booleana* es una expresión que, al evaluarse, devuelve un valor de tipo *Booleano*. El caso más sencillo de expresión *booleana* es, simplemente, una variable que almacene un valor verdadero (*true*) o falso (*false*). En el siguiente apartado se verán algunos ejemplos de expresiones y se abundará en este concepto.

Un ejemplo de sentencia *if* es el siguiente (obsérvese la tabulación):

```
if festivo then
        descansar;
        comer;
        dormir;
end if;
```

descansar, *comer* y *dormir* son tres procedimientos cualesquiera, cuyo nombre se ha escogido con fines ilustrativos. En Ada, no es necesario que la llamada a un procedimiento o función que no acepte parámetros de entrada, vaya acompañada de dos par_ntesis, como

descansar(), tal y como ocurre en la mayoría de los lenguajes. Si la función o procedimiento necesitara parámetros, sí que se especificarían los mismos dentro de un paréntesis, como en *calcula_cuadrado(X)*. Funciones, procedimientos y paquetes serán objeto de amplio estudio en próximos capítulos.

En el ejemplo, se supone que *festivo* es una variable de tipo *Booleano*. También podría ser una función que devolviera un valor de este tipo. Si dicho valor fuera *True* (recuérdese del capítulo anterior que el tipo *Booleano* sólo tenía dos valores posibles, *True* y *False*, "cierto" y "falso") se ejecutarían las sentencias contenidas entre las palabras *then* y *end if;* en caso contrario, es decir, si el valor fuera *False*, no se ejecutaría nada. Si se quisiera especificar, en este último caso, una secuencia alternativa de sentencias, se puede usar la palabra reservada *else* tal y como aparece en el siguiente ejemplo.

```
if festivo then
        descansar;
        comer;
        dormir;
else
        trabajar;
        tomarCafe;
        trabajar;
end if;
```

Dentro de un bloque *if* puede aparecer cualquier sentencia. Por lo tanto, es posible anidar varias sentencias *if*, unas dentro de otras. Es muy recomendable indentar con un tabulador el código de forma que se vea claro el nivel de anidamiento en el que se encuentra cada sentencia. Con ello se consigue que el código sea más claro y legible:

```
if A>0 then
        ó aquí vendrian algunas sen-
tencias
        if A>10 then
                o aquí otras
        else
                o aquí otras más
        end if;
end if;
```

En este ejemplo se supone que *A* es una variable de tipo entero. La expresión *booleana A>0* toma el valor

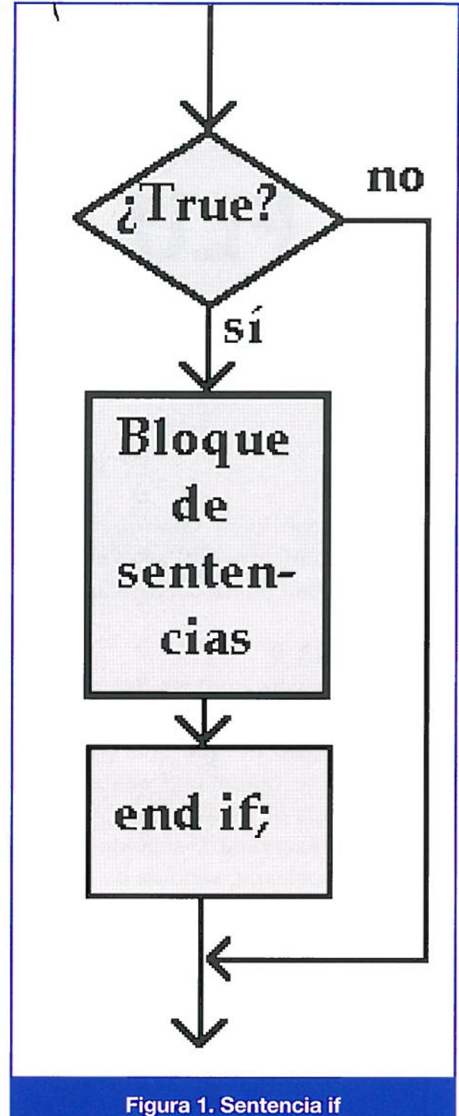


Figura 1. Sentencia if

True cuando el contenido de A es mayor que 0.

Una construcción bastante habitual en los programas tradicionales es la concatenación de sentencias *if ... else if*, para escoger una de entre varias posibilidades:

```
if ... then
        else if ... then
        ...
                else if ... then
                ...
                        end if;
                end if;
end if;
```

Lo cierto es que el código resultante resulta bastante poco elegante, puesto que se están anidando las distintas alternativas, cuando en realidad éstas son mutuamente excluyentes y están, "conceptualmente", al mismo nivel. Como efecto secundario, meramente estético, debido a la recomendación de indentar cada nivel de anidamiento el código se va desplazando hacia la derecha.

Para evitar estos inconvenientes, en Ada se dispone de la palabra reservada *elsif*, que nos permite reescribir la construcción anterior de forma más compacta:

```
if ... then
        o aquí vendrían algunas sen-
tencias
elsif ... then
        o y aquí otras
elsif ... then
        o y aquí otras
else
        o y aquí otras
end if;
```

Se puede comprobar que con el uso de *elsif*, no existe anidamiento, y sólo debe aparecer un *end if*.

EXPRESIONES BOOLEANAS

El tipo *Boolean* es un tipo enumeración cuya definición se vio en el capítulo anterior, dedicado a tipos de datos. Se recuerda a continuación:

```
type Boolean is (False, True);
```

Una forma de obtener un valor *Booleano* es mediante los operadores =, /= ("distinto de"), <, <= ("menor o igual que"), >= y >. Dichos operadores pueden actuar sobre muchos tipos distintos. Sirven para comparar parejas de valores. Su significado es evidente en tipos numéricos, que es lo más habitual. Existen otros operadores que actúan sobre el tipo *Booleano* que permiten construir expresiones más complejas que las simples comparaciones. Ellos son *not, and, or* y *xor*.

El operador *not* es "unario". Actúa sobre una sola expresión. Nótese que no es necesario que la expresión sobre la que actúa el operador vaya entre paréntesis. Los demás operadores son binarios, y relacionan por tanto dos expresiones *booleanas*.

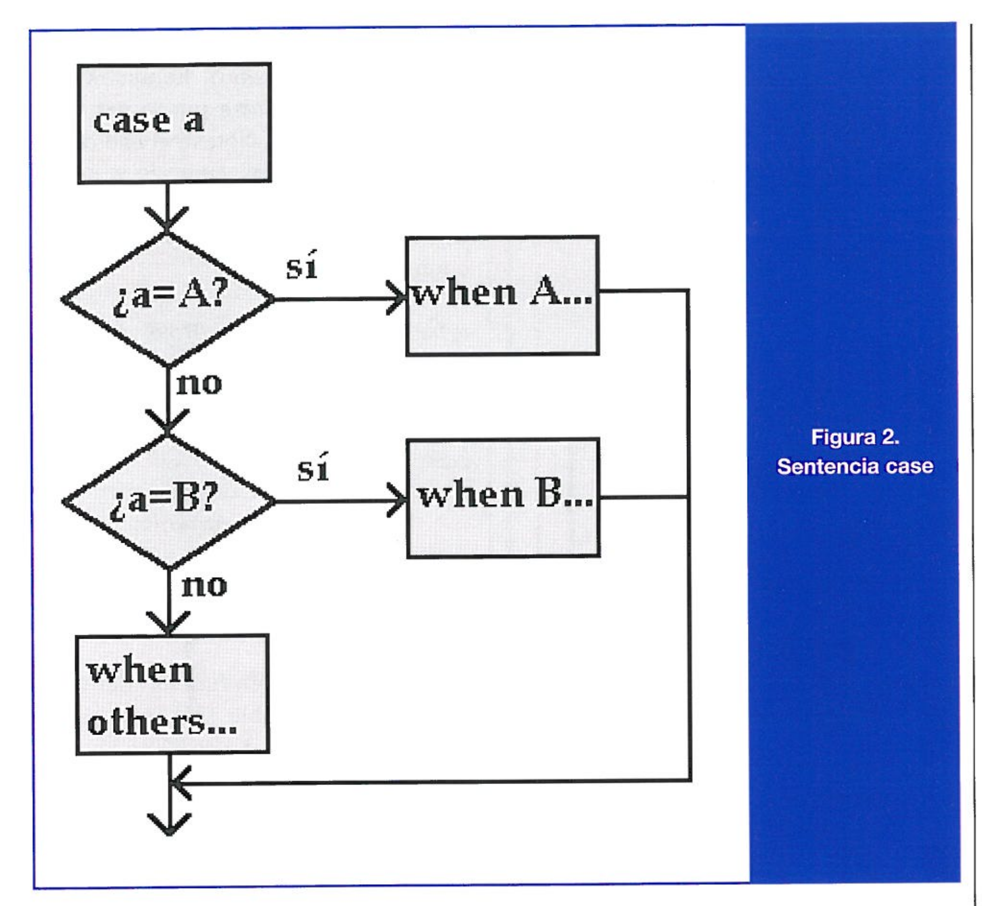


Figura 2. Sentencia case

Todos estos operadores se describen en la tabla adjunta.

not a
Actúa sobre una expresión *Booleana* y devuelve la negación de la misma. Es decir, *not True* es igual a *False* y *not False* es igual a *True.*
a and b Devuelve el valor *True* cuando *a* y *b* son ambos de valor *True.*
Devuelve *False* en caso contrario.
a or b Devuelve el valor *True* en el caso de que *a* o *b* sean, al menos uno de ellos, de valor *True.*
Devuelve *False* en caso contrario.
a xor b Devuelve el valor *True* cuando *a=not b.*

Devuelve *False* en caso de que *a=b.*

La precedencia de los operadores *and*, *or* y *xor* es menor que la del resto de los operadores del lenguaje. Esto quiere decir que

A=B and C=D

es lo mismo que

(A=B) and (C=D)

Por otra parte, la mayor precedencia del operador *not* hace que

not A and B

se evalúe como

(not A) and B

y no como

not (A and B)

Conviene llamar la atención sobre otros dos detalles relacionados con las expresiones *Booleanas*. Para ello, supóngase que *E* es una variable de tipo *Boolean*. Es decir,

E : Boolean;

La expresión *(E=true)* es exactamente igual a *(E)*. Por lo tanto, la sentencia

```
if E=true then
        o sentencias
end if;
```

y la sentencia

```
if E then
        o sentencias
end if;
```

son equivalentes, dado que *E* ya es en sí misma una variable *Boolean*. Por otra parte, si *A* es una variable de tipo *Integer*, entonces

E:= A=10;

es lo mismo que

```
if A=10 then
        E=True;
else
        E=False
end if;
```

excepto que esta segunda forma es innecesariamente más larga. Con un poco de práctica, el lector se habituará a la forma de estas expresiones.

El orden en que se evalúan las expresiones conectadas por los operadores *or* y *and* se deja, por defecto, a elección del compilador, por razones de optimización del código. Así, al contrario que otros lenguajes, un compilador sagaz puede elegir en la sentencia *if* siguiente:

```
if ( funcLenta ) or ( funcRapida ) then
        ...
end if;
```

evaluar primero la expresión que menos tiempo tarde, y en caso de que sea falsa y SÓLO en ese caso, ha de evaluar la lenta.

Por otra parte, en una expresión del tipo *(A and B)*, si el compilador decide evaluar primero *A* y resulta que obtiene un valor *False*, entonces puede prescindir de evaluar *B*, dado que ya sabe que *(A and B)* va a resultar ser *False*. Esto también es una forma de optimizar el código, puesto que lo hace más rápido.

Sin embargo, en ciertas circunstancias, es necesario especificar al compilador cuál de las dos expresiones se debe evaluar primero. Ello puede ocurrir cuando la evaluación de la segunda expresión, en caso de que la primera fuera falsa, produzca un error.

Hay que garantizar, por ello, que la segunda expresión se evalœa sólo cuando la primera sea cierta. A continuación se describe una situación que aclarará el problema que se intenta presentar:

```
X,Y: Integer;
...
if X/=0 and Y/X=2 then
        o sentencias
end if;
...
```

Si el valor de *X* en el momento de ejecutar la sentencia *if* fuese 0, entonces la decisión del compilador de evaluar primero *X/=0* o *Y/X=2* puede conducir a resultados muy diferentes. Si eligiese evaluar primero *X/=0*, obtendría *False*, y por lo tanto ya no comprobaría *Y/X=2*, dado que no es necesario. El programa no ejecutaría el bloque de sentencias señalado en el ejemplo y continuaría tranquilamente. Sin embargo, si el compilador decidiese ejecutar *Y/X* en primer lugar, entonces se levantaría una excepción (un error) indicando un intento de división por cero. El programa se detendría. Una solución correcta a este problema consistiría en dividir la sentencia *if* en dos sentencias anidadas:

```
if X/=0 then
        if Y/X=2 then
                o sentencias
        end if;
end if;
```

Sin embargo, Ada permite una solución más cómoda mediante el uso de las palabras reservadas *and then* en lugar de únicamente *and*:

```
if X/=0 and then Y/X=2 then
        o sentencias
end if;
```

Con ello se obliga al compilador a que compruebe *X/=0* y sólo en el caso de que esta expresión resulte ser cierta, entonces comprobaría si *Y/X=2*.

Un problema similar al descrito ocurre con el operador *or*. Supóngase que se tiene una expresión como *(A or B)*. Si el compilador decidiera evaluar *A* en primer lugar, y si al hacerlo, obtuviera el valor *True*, ya sabría que la expresión completa *(A or B)* resultará ser cierta, independientemente del valor de *B*. Por ello, no malgastará su tiempo en evaluar *B*. Si, por motivos análogos a los presentados en el caso del operador *and*, fuera necesario evaluar *A* antes que *B*, de forma que *B* sólo se evaluase en el caso de que *A* fuera *False*, ello se podría indicar al compilador con las palabras reservadas *or else*, en vez de únicamente el operador *or*:

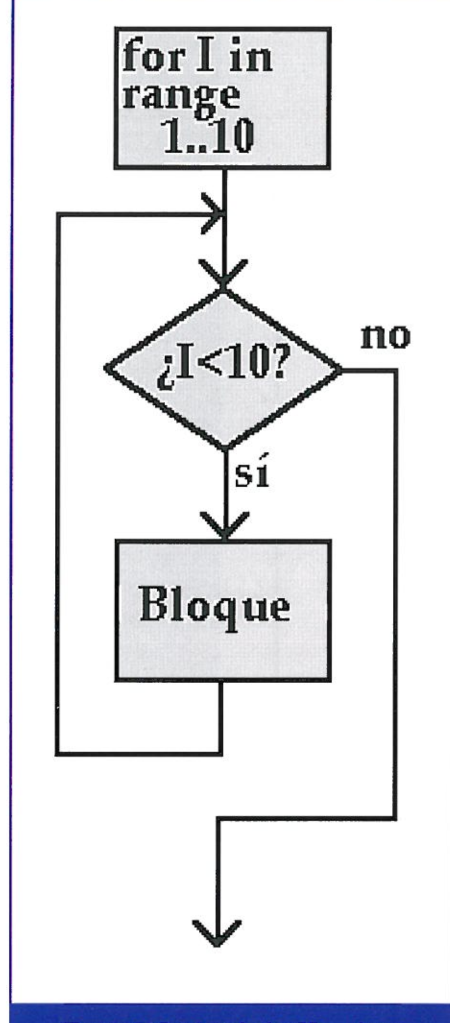


Figura 3. Sentencia for I in ... loop

```
if X=0 or else Y/X=2 then
        o sentencias
end if;
```

SENTENCIA CASE

La sentencia *case* sirve para elegir la ejecución de uno entre varios bloques de sentencias en función del valor que adopte una variable (o, más genéricamente, una expresión). La sintaxis de una sentencia *case* se puede ver en el siguiente ejemplo. Nótese el uso de las palabras reservadas *case*, *is* y *when*, así como del símbolo => (que no debe confundirse con el operador >= "mayor o igual que").

```
type Luz_Semaforo is (Rojo, Amarillo, Verde);
Luz : Luz_Semaforo;
...
case Luz is
        when Rojo => Detenerse;
        when Amarillo => AcelerarMucho;
        when Verde => Acelerar;
end case;
```

En el ejemplo, en función del valor que tome la variable *Luz* (que puede ser *Rojo*, *Amarillo* o *Verde*) se invocará uno de los tres procedimientos indicados (respectivamente, *Detenerse*, *AcelerarMucho* o *Acelerar*). El lector habrá observado que los dise–adores de Ada han puesto mucho esfuerzo en hacer que la sintaxis de Ada sea lo más parecida al inglés posible.

Ada nos obliga a especificar una alternativa para cada posible valor que tome la variable o expresión. Si hay varios valores para los que la acción a ejecutar sea la misma, _stos se pueden agrupar usando el símbolo |. Tambi_n, para facilitar esta misma tarea, se puede hacer uso de rangos.

Es posible utilizar la palabra reservada *others* para referirse a todos los demás valores de la variable que no hayan sido contemplados explícitamente. Por último, se debe tener en cuenta que todas las alternativas de una sentencia *case* deben elaborarse de forma estática, en tiempo de compilación. Esto significa que no pueden ser expresiones que contengan variables, ni llamadas a funciones.

Otro ejemplo, más completo, del uso de una sentencia *case* es el siguiente:

```
type Calificacion is Integer range 0..10;
Examen : Calificacion;
...
case Examen is
        when 0..4 => Estudiar;
```




```
        when 9|10 => Celebrar;
        when others => Descansar;
end case;
```

En caso de aparecer, *others* debe ponerse como la última posibilidad y no puede combinarse con ningún otro valor. *others* se usa para especificar las sentencias que se ejecutan "por defecto". En el ejemplo, *others* hace referencia a los valores de examen comprendidos entre el 5 y el 8, inclusive.

En una estructura *case* se puede hacer uso de la sentencia *null*. La sentencia *null*, que también aparecerá en otros contextos, es una sentencia especial que sirve para indicar al compilador que no se quiere realizar ninguna operación. Existe para que se pueda expresar, explícitamente, el deseo de no hacer nada. Por ejemplo,

```
case Examen is
        when 0..4 => estudiar;
        when 9|10 => Celebrar;
        when others => null;
end case;
```

La sentencia case sirve para ejecutar uno de entre varios bloques de código en función del valor que adopte una expresión

BUCLES (SENTENCIAS LOOP)

En Ada existen varias formas de ejecutar un bloque de sentencias de manera repetida. La más sencilla hace uso de la sentencia *loop* únicamente y sirve para ejecutar las sentencias indefinidamente. Por ejemplo, un sistema de control que tuviese que leer continuamente datos de un sensor y actuar en función de dichos datos, podría tener un bucle principal como el siguiente:

```
loop
        Datos=LeerSensor;
        ControlarSistema(Datos);
end loop;
```

Se puede abandonar el interior de un bucle mediante la sentencia *exit*. Cuando se ejecuta una sentencia de este tipo, el programa continúa su ejecución después del punto donde aparece *end loop*. En el caso de que *exit* aparezca en el interior de varios bucles anidados, se sale únicamente del bucle interior (aunque se puede especificar que no sea así, como se verá más adelante). *Exit* se utiliza en construcciones del tipo: *if condición then exit; end if;* De hecho, Ada proporciona una forma más compacta para escribir lo anterior. Es mediante la construcción *exit when condición;* Por ejemplo,

```
loop
        Luz=MirarSemaforo;
        exit when Luz=Verde;
end loop;
```

Como ya se ha mencionado antes, es posible, en el caso de tener varios bucles anidados, abandonar todos los bucles de la construcción anidada con una sola sentencia *exit*. Para ello es necesario nombrar el bucle más exterior que se quiere abandonar y hacer referencia a él en la sentencia *exit*. Para nombrar un bucle, se precede la palabra reservada *loop* del nombre que se quiera dar al bucle, acabado en dos puntos. Adicionalmente, es necesario terminar el bucle así nombrado especificando su nombre detrás de su *end loop*.

```
Bucle_exterior:
loop
        loop
                ...
                exit Bucle_exterior when condicion;
                ...
        end loop;
end loop Bucle_exterior;
```

Se ha visto cómo realizar bucles que se repiten de forma indefinida. Si se quiere que el bucle se repita en tanto que una determinada condición sea cierta, se hace uso de la palabra reservada *while*. *while* se antepone a la expresión *booleana* que se va a evaluar en cada iteración. Dicha expresión va seguida del bloque *loop <sentencias> end loop*, que tiene el mismo aspecto que antes. Cada vez que va a iniciarse un nuevo recorrido por las sentencias del bucle, se comprueba la veracidad de la expresión. En el caso de que dicha expresión resulte *False*, se continúa la ejecución detrás de *end loop*. Como caso particular, si la primera vez que se va a ejecutar el bucle, la expresión resulta ser falsa, las sentencias en el interior del bucle no se ejecutarán ninguna vez. Un ejemplo de bucle usando *while* es el siguiente:

```
declare
        I:Integer :=1;
begin
        while I<100 loop
                I=I+1;
        end loop;
end;
```

Es bastante habitual el caso de tener una variable de tipo discreto (por ejemplo, de tipo *Integer*) que actúe como índice del bucle y que lleve una cuenta de las veces que el bucle se ha ejecutado. En el ejemplo anterior, la variable I jugaba este mismo papel. De esta forma, se puede controlar el número exacto de veces que se va a repetir un bucle. Ada dispone, para ello, de otro tipo especial de bucle, el bucle *for*. Un bucle *for* tiene la siguiente apariencia:

```
for I in 1..100 loop
```

En Ada también existen los bucles while y for que están presentes en otros lenguajes

```
        ...
end loop;
```

El índice (en el ejemplo, *I*) es una variable que no existe fuera del bucle y no necesita estar declarada fuera de él. Puede ser de cualquier tipo discreto. El tipo al que pertenece viene determinado por el rango que aparece en la sentencia *for* (en el ejemplo, *I* es un tipo entero). En caso de ambigüedad, por ejemplo porque el rango se pueda referir a dos tipos enumeración distintos, se puede especificar el tipo explícitamente:

```
for Luz in Luz_Semaforo range
Rojo..Verde loop
        ...
end loop;
```

o bien, un poco peor,

```
for Luz in Luz_Semaforoí(Rojo)..Verde
loop
        ...
end loop;
```

Los límites del rango no necesitan ser calculados estáticamente (esto es, en tiempo de compilación), sino que pueden depender de otras variables o llamadas a funciones. Existen varias particularidades en el comportamiento del índice de un bucle. La primera es que los límites del bucle sólo se evalúan una vez, al entrar en el bucle, y no pueden ser alterados desde el interior del mismo. Por ejemplo, el bucle

```
X:=10;
for I in 1..X loop
        ...
        X:=20;
end loop;
```

se evalúa 10 veces, aunque el valor de *X* se cambie a 20 dentro de él. Por otra parte, si se da la circunstancia de que *X* sea menor que 1, el rango así definido *(1..X)* se supone nulo y el bucle no se ejecuta ninguna vez. Esto ya se señaló en el capítulo dedicado a tipos.

El índice de un bucle for tiene unas propiedades muy particulares

Otro aspecto importante del índice es que, al existir sólo dentro del bucle, su valor "final" no puede ser leído una vez finalizado el mismo. Más importante es el detalle de que en el interior del bucle, el índice se comporta como una constante. Su valor no puede ser alterado excepto por el propio mecanismo del bucle, que incrementa (o decrementa, como se verá en seguida) su valor en una unidad en cada iteración. No se puede especificar un paso distinto de uno. En realidad, estas restricciones no sacrifican generalidad y obligan a un código más claro. En concreto, se puede saber el nœmero de veces que se va a repetir un bucle con mirar tan sólo la cabecera.

Sí se permite, en cambio, elegir que el rango se recorra en sentido ascendente o descendente. Esto último se hace intercalando la palabra reservada *reverse* antes de especificar el rango (*for I in reverse 1..100 loop*). Nótese que el rango siempre se define en orden ascendente.

SENTENCIA GOTO

goto es una sentencia que sirve para transferir el control a cualquier otra parte del procedimiento o función en el que se encuentre la ejecución. El punto al que se quiere saltar debe marcarse con una etiqueta, y en ningœn caso puede estar en el interior de una sentencia *if*, un bucle, o una sentencia *case*. Una etiqueta es un marcador, un nombre, que señala un punto del programa. Debe encerrarse entre dobles llaves, << >>, tal y como se presenta a continuación:

```
...
<<Etiqueta>>
...
goto Etiqueta;
...
```

Se recomienda no hacer uso nunca de sentencias *goto*, puesto que oscurecen enormemente el significado del código. No es casualidad que la definición de etiquetas tenga una sintaxis tan llamativa. El uso de los demás mecanismos de control debe ser más que suficiente para cubrir todas las necesidades que puedan surgir en el control de flujo sin tener que recurrir al *goto*. En lenguajes más antiguos, el uso del *goto* sólo se justificaba en el caso de tener que sacar la ejecución desde el interior de una estructura con muchos bucles anidados. En Ada se puede realizar esto mediante la sentencia *exit*, tal y como se ha visto, o bien levantando una excepción. Este œltimo procedimiento se verá en el tan anunciado capítulo dedicado a excepciones.

La pregunta que el lector se puede hacer es: por qué existe, entonces, la sentencia *goto*. La respuesta es que dicha sentencia facilita la tarea de algunos generadores automáticos de código Ada. Un ejemplo de esto puede ser un traductor de programas de otros lenguajes a Ada o un programa que genere código Ada a partir de alguna especificación de más alto nivel.

CONCLUSIÓN

Este mes se han revisado los mecanismos de control de flujo secuencial que proporciona Ada. Estos son las sentencias *if*, las sentencias *case*, y los diferentes tipos de bucles. Por último se ha hecho mención a la sentencia *goto*, haciendo hincapié en que el mejor uso de dicha sentencia es no usarla en absoluto.

En la parte más avanzada de este curso se verán otros mecanismos de control de flujo, no secuenciales, que nos permitirán crear varios flujos de ejecución para que nuestro programa realice varias tareas al mismo tiempo (de forma concurrente, en el sentido que se indicó al principio del capítulo). Esta parte constituye uno de los puntos fuertes de Ada y resulta ser un elemento indispensable en el diseño de sistemas de control de tiempo real.

CONCEPTOS CLAVE: flujo de ejecución, sentencias *if* y *case*, bucles *loop*, *while* y *for* y sentencia *exit*

CONTROL DEL RATÓN Y EL TECLADO

Juan Manuel y Luis Martín

Además de los eventos relacionados con los clics del ratón, que ya han sido explicados en artículos anteriores, los formularios y controles de Visual Basic disponen de otros eventos que les permiten detectar acciones del ratón aún más elementales. Así, existen eventos que se producen separadamente tanto al pulsar como al liberar un botón, al situar el ratón sobre un objeto, al arrastrar un objeto por una superficie, etc.

Además, Visual Basic dispone de una serie de eventos que, al igual que ocurre con el ratón, permiten detectar separadamente las distintas pulsaciones y liberaciones de teclas realizadas durante la ejecución. De esta forma, es posible realizar aplicaciones o partes de ellas controladas por el teclado.

LOS EVENTOS DEL RATÓN

En artículos anteriores se ha podido comprobar que las principales acciones que pueden realizarse con el ratón son el clic y el doble clic sobre los objetos. En respuesta a estas acciones, se producen respectivamente los eventos Click y DblClick, que forman parte del repertorio de la mayoría de los objetos, aunque no de todos.

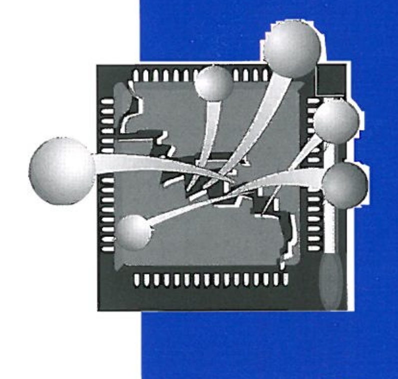
Para distinguir entre un doble clic y dos clics realizados de forma consecutiva, Windows utiliza un intervalo de tiempo prefijado, cuyo valor puede establecerse en el Panel de Control de Windows. Este intervalo se compara con el tiempo transcurrido entre ambas pulsaciones. Si el tiempo entre pulsaciones es inferior al intervalo prefijado, Windows interpreta que se ha realizado un doble clic, generando el evento correspondiente. Sin embargo, si dicho tiempo es superior al intervalo prefijado, Windows interpreta que se han realizado clics simples, generando el evento correspondiente para cada uno de ellos.

Aunque los eventos Click y DblClick permiten realizar un cierto control sobre las pulsaciones del ratón, existen otros dos eventos que permiten realizar el control de forma mucha más exhaustiva. Se trata de los eventos MouseDown y MouseUp, que se producen al pulsar y liberar un botón del ratón, respectivamente. Estos eventos son detectados por el formulario o control sobre el que se encuentra el puntero del ratón en ese momento. En el caso de los formularios, para que sean detectados estos eventos, el puntero del ratón debe encontrarse sobre un zona del mismo libre de controles.

Algunos controles no disponen de estos eventos, ya que su detección podría entrar en conflicto con el propio funcionamiento del control. Así, por ejemplo, en las barras de desplazamiento no sería posible distinguir entre una pulsación arbitraria y una realizada para llevar a cabo una operación de desplazamiento. Los formularios MDI y los controles Line y Shape tampoco disponen de estos eventos.

A la hora de manejar estos eventos, es importante tener en cuenta el orden en que se producen. Cuando se realiza un clic sobre un objeto, el orden en que se producen los eventos es: MouseDown, MouseUp y Click. Por otro lado, cuando se realiza un doble clic, el orden es: MouseDown, MouseUp, Click, DblClick y un segundo MouseUp.

Sin embargo, esto no siempre es así, ya que el usuario puede realizar la pulsación sobre un objeto y, manteniendo pulsado el botón del ratón, arrastrar el puntero hasta otro objeto, liberando



Tanto el propio Windows como la mayoría de las aplicaciones que trabajan bajo éste están pensadas para ser utilizadas con un dispositivo apuntador como es el ratón. Los movimientos y los *clics* de éste permiten controlar el funcionamiento de las aplicaciones de forma sencilla e intuitiva. Sin embargo, en la mayoría de los casos sigue siendo necesario el teclado como dispositivo para la introducción de datos por parte del usuario.

entonces el botón. En este caso, el objeto sobre el que se realiza la pulsación detectaría los eventos MouseDown y MouseUp, pero no el evento Click. El objeto sobre el que se libera el botón no detectaría ningún evento. El cuadro 1 contiene un ejemplo que ilustra el funcionamiento de estos eventos, permitiendo al lector experimentar con diversas combinaciones de pulsaciones y liberaciones.

Como ha podido comprobarse en el ejemplo anterior, sólo el objeto sobre el que se encuentra el ratón en el momento de la pulsación puede recibir estos eventos. En general, suele decirse que dicho objeto tiene capturado el ratón. De esta forma, cuando la pulsación se realiza en un control situado sobre un formulario, los eventos no son detectados por el formulario, ya que el control tiene capturado el ratón.

La sintaxis de los procedimientos de evento MouseDown y MouseUp puede resumirse de la siguiente forma:

```
Private Sub Objeto_MouseDown ([Index As Integer],Button As Integer,
  Shift As Integer, X As Single, Y As Single)

Private Sub Objeto_MouseUp([Index As Integer],Button As Integer,
  Shift As Integer, X As Single, Y As Single)
```

El argumento Button indica el botón del ratón que ha sido pulsado o liberado. Para ello, devuelve un entero cuyos bits 0, 1 y 2 se encuentran activos en función del botón. Así, un valor 1 indica que se ha pulsado o liberado el botón izquierdo, un valor 2 el botón derecho y un valor 3 el botón central (en caso de existir). La librería VB dispone de constantes intrínsecas definidas para estos tres valores: vbLeftButton, vbRightButton y vbMiddleButton.

El argumento Shift indica el estado de las teclas [Shift], [Control] y [Alt] en el momento de producirse el evento. Al igual que el argumento anterior, devuelve un número entero cuyos bits 0, 1 y 2 se encuentran activos en función de la tecla. Así, un valor 1 indica que se encuentra pulsada la tecla [Shift], un valor 2 que se encuentra pulsada la tecla [Control] y un valor 3 que se encuentra pulsada la tecla [Alt]. La librería VB también dispone de constantes intrínsecas definidas para representar estos tres valores: vbShiftMask, vbControlMask y vbAltMask.

Sin embargo, y a diferencia del argumento anterior, es posible que en el momento de producirse el evento se encontrara pulsada más de una tecla o ninguna. Cuando no se encuentra pulsada ninguna tecla el valor devuelto por el argumento es 0, mientras que si se encuentra pulsada más de una tecla el valor devuelto será la suma de los valores correspondientes a cada tecla. Así, por ejemplo, si se encontraban pulsadas las teclas [Shift] y [Alt], el valor devuelto será 4, es decir, vbMaskShit + vbMaskAlt.

Para comprobar si una de las teclas anteriores se encontraba pulsada en el momento de producirse el evento, pueden utilizarse las constantes intrínsecas en forma de máscaras de bit. Para ello, basta realizar una operación And entre el argumento Button y la constante de máscara correspondiente, verificando si el resultado es positivo.

```
If Button And vbControlMask > 0 Then
Form1.Print "Se encuentra pulsada la tecla [Control]"
End If
```

Finalmente, los argumentos X e Y indican la posición del puntero del ratón en el momento de producirse el evento, referida a la esquina superior izquierda del objeto. Sus valores vienen determinados por los valores actuales de las propiedades de escala del objeto. Estos valores realmente representan el desplazamiento u offset del puntero del ratón respecto del control.

En el caso del evento MouseDown, las coordenadas X e Y estarán siempre restringidas al espacio del control. Sin embargo, al liberar el botón fuera del espacio del control, las coordenadas devueltas por MouseUp pueden quedar fuera de dicho rango, pudiendo ser incluso negativas si se libera en la parte superior izquierda del control.

Del mismo modo que ocurre con las pulsaciones de los botones del ratón, los movimientos de éste también pueden ser controlados. Para ello, se utiliza el evento MouseMove. Se trata de un evento detectado por el formulario o control sobre el que se encuentra situado el puntero del ratón, y que se produce cada vez que éste se mueve. Como siempre, para ser detectado por un formulario, el puntero del ratón debe encontrarse situado sobre una zona de éste libre de controles.

La sintaxis del procedimiento de evento correspondiente es muy similar a la de los eventos MouseDown y MouseUp:

```
Private Sub Objeto_MouseMove ([Index As Integer],Button As Integer,Shift As Integer, X As Single, Y As Single)
```

El significado de los distintos argumentos es idéntico al que se ha explicado para los eventos MouseDown y MouseUp, excepto en el caso del argumento Button. El argumento Button de MouseMove actúa también como una máscara, de forma que es posible que se devuelva un valor 0 cuando no se encuentra pulsado ningún botón, o una combinación de valores si se encuentra pulsado más de un botón.

El evento MouseMove se produce cada vez que se mueve el puntero del ratón sobre un formulario o control

El siguiente ejemplo ilustra el funcionamiento del evento MouseMove, permitiendo dibujar líneas sobre la superficie del formulario. Para ello, bastará con incluir en el formulario por defecto el siguiente procedimiento de evento:

```
Private Sub Form_MouseMove (Button As Integer, Shift As Integer,
 X As Single, Y As Single)
 If Button > 0 Then
       Form1.Line  (Form1.CurrentX,
```

CUADRO 1.

1. Añadir un control cuadro de dibujo (PictureBox) al formulario por defecto, situándolo en la mitad derecha del mismo.

2. Escribir los siguientes procedimientos de evento:

```
Private Sub Picture1_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
Picture1.Print "Evento MouseUp del control"
End Sub

Private Sub Picture1_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
Picture1.Print "Evento MouseDown del control"
End Sub

Private Sub Form1_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
Form1.Print "Evento MouseUp del formulario"
End Sub

Private Sub Form1_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
Form1.Print "Evento MouseDown del formulario"
End Sub
```

3. Salvar el formulario en el archivo MOUSE.FRM y el proyecto en el archivo MOUSE.VBP.

Ejemplo ilustrativo de los eventos MouseUp y MouseDown.

```
Form1.CurrentY)-(X, Y)
  End If
End Sub
```

Como puede observarse, el ejemplo anterior dibuja una línea que une los distintos puntos por los que pasa el puntero del ratón, siempre y cuando se encuentre pulsado alguno de sus botones.

OPERACIONES DE ARRASTRE

Una de las acciones típicas de las aplicaciones Windows es la posibilidad de mover objetos mediante operaciones "arrastrar y colocar" (Drag & Drop). Esto es posible gracias a la naturaleza orientada a objetos de Windows. De esta forma, es posible mover un objeto con el ratón y soltarlo en otra posición. Para ello, debe situarse el puntero del ratón sobre el objeto que se desea mover y pulsar un botón del ratón. Manteniendo pulsado el botón, es posible mover el objeto con el ratón. Una vez en la posición deseada, se libera el botón del ratón.

Visual Basic también soporta estas operaciones. Para ello, dispone de métodos y eventos especializados en su manejo. Sin embargo, previamente es necesario aclarar que en este tipo de operaciones se ven involucrados siempre dos objetos:

- Objeto Fuente: Es el objeto que se arrastra (Drag) con el ratón.
- Objeto Destino: Es el objeto sobre el que se suelta (Drop) el objeto fuente.

Para manejar el objeto fuente en una operación de arrastre se utiliza un método que permite activar o desactivar el proceso. Por el contrario, el objeto fuente se maneja mediante dos eventos que indican si el objeto fuente ha sido soltado sobre él o simplemente ha pasado por encima.

Para arrastrar un control se utiliza el método Drag. Se trata de un método que simplemente activa, desactiva o cancela el proceso de arrastre. Cuando se activa el proceso de arrastre de un control, sobre los bordes del mismo se visualiza un recuadro móvil del mismo tamaño que el control, en un color inverso al normal. Tanto el recuadro móvil como el puntero del ratón pasarán a moverse simultáneamente. La sintaxis del método es la siguiente:

```
Objeto.Drag [Acción]
```

En el argumento Acción debe especificarse un valor numérico que indica el tipo de operación a realizar. Un valor 1 (vbBeginDrag) permite especificar el comienzo de la operación de arrastre (Dragging). Un valor 2 (vbEndDrag) permite especificar el final de la operación de arrastre (Dropping). Finalmente, un valor 0 (vbCancel) indica que la operación de arrastre ha sido cancelada. Si se omite el argumento, el valor por defecto es 1.

Normalmente, el proceso de arrastre suele iniciarse al pulsar un botón del ratón sobre el objeto fuente. Para ello, debe incluirse la llamada al método Drag en su procedimiento de evento MouseDown. Sin embargo, es posible iniciar esta operación con cualquier otra acción.

También es posible cambiar el tipo de representación del control que está siendo arrastrado, que por defecto es el puntero en forma de flecha, dentro de un recuadro con las mismas dimensiones que el control. Para ello, debe utilizarse la propiedad DragIcon de dicho objeto. Esta propiedad permite especificar un icono que será visualizado como puntero durante la operación de arrastre. La asignación de valores puede realizarse a partir de las propiedades Icon y DragIcon de otro objeto, o utilizando la función LoadPicture. También es posible consultar si esta propiedad no tiene asignado un valor. En este caso, el valor devuelto será False.

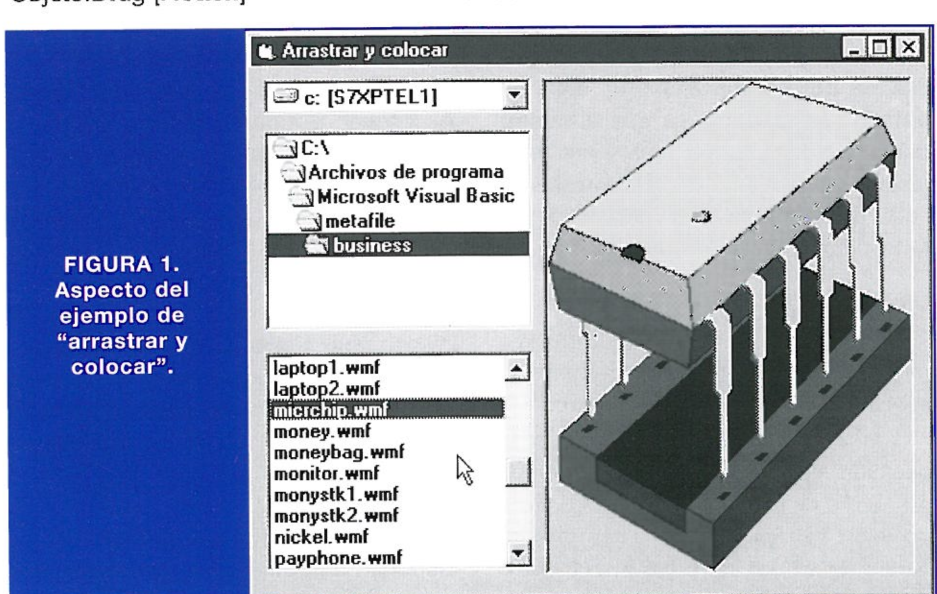


FIGURA 1. Aspecto del ejemplo de "arrastrar y colocar".

```
If Image1.DragIcon = False Then
                Image1.DragIcon =
LoadPicture("C:\VB4\ICONS\OFFICE\FILES01A.ICO")
Else
  Image1.DragIcon = LoadPicture()
End If
```

La operación "arrastrar" no tiene sentido si no viene acompañada de una operación "soltar". Una vez que el objeto fuente ha sido arrastrado y situado sobre el destino deseado debe soltarse. En ese momento, termina la operación de arrastre y se produce el evento DragDrop asociado al objeto destino. El procedimiento de evento correspondiente es el que debe utilizarse para definir las acciones para las que se realizó la operación de arrastre. La sintaxis de este procedimiento de evento es la siguiente:

```
Private Sub Objeto_DragDrop ([Index As Integer,] Source As Control,
  X As Single, Y As Single)
```

El argumento Source contiene un valor de tipo control que indica el control fuente que ha sido soltado sobre el objeto. Este argumento es una referencia al control que se ha soltado, permitiendo especificar propiedades e invocar métodos del mismo. Sin embargo, debe ser tratado con cuidado, ya que el control puede ser de cualquier tipo, y puede no tener la propiedad o el método especificado, generándose así un error. Para evitarlo, debe utilizarse el operador TypeOf.

```
If TypeOf Source Is Image Then
Source.Picture = LoadPicture
("C:\VB4\BITMAPS\IMAGEN.BMP")
End If
```

Los argumentos X e Y indican la posición del puntero del ratón en el momento de producirse el evento. Esta posición vendrá referida a las propiedades de escala del objeto destino, o de su contenedor.

TABLA 1.

Tecla	Valor de KeyAscii
Caracteres Imprimibles	Código ANSI correspondiente
De [Ctrl+A] hasta [Ctrl+Z]	1 - 26
[Retroceso]	8
[Tab]	9
[Ctrl+Intro]	10
[Intro]	13
[Ctrl+Retroceso]	127

Teclas que producen el evento KeyPress.

Cuando se realiza una operación de arrastre para cambiar la posición del control, es importante tener en cuenta que los argumentos X e Y se refieren al puntero del ratón, y no a la esquina superior izquierda del recuadro móvil. Por tanto, al soltar el objeto, será necesario ajustar su posición para que ésta coincida con la del recuadro móvil. Así, para obtener la nueva posición del control será necesario sustraer a estos argumentos la distancia del puntero del ratón a la esquina superior izquierda del recuadro móvil. Para ello, será necesario almacenar dicha distancia en el momento de producirse el evento MouseDown. Así, por ejemplo, para mover una imagen dentro de un formulario, puede utilizarse el siguiente código:

```
Private Sub Image1_MouseDown (Index As Integer, Button As Integer,
  Shift As Integer, X As Single, Y As Single)
  DistX = X
  DistY = Y
  Image1.Drag
End Sub
Private Sub Form_DragDrop(Index As Integer, Source As Control,
  X As Single, Y As Single)
  Source.Move (X - DistX), (Y - DistY)
End Sub
```

Los objetos de Visual Basic también disponen de otro evento que se produce cuando el objeto arrastrado pasa sobre ellos. Se trata del evento DragOver cuya sintaxis es la siguiente:

```
Private Sub Objeto_DragOver ([Index As Integer], Source As Control,
X As Single, Y As Single, State As Integer)
```

Los parámetros Source, X e Y tienen el mismo significado que en el caso del procedimiento de evento DragDrop. El parámetro State contiene un valor entero que indica el tipo de transición realizada por el control que está siendo arrastrado respecto del objeto. Así, un valor 0 (vbEnter) indica que el control fuente está entrando en la superficie del objeto. Un valor 1 (vbLeave) indica que está saliendo. Finalmente, un valor 2 (vbOver) indica que el control fuente se está moviendo dentro de la superficie del objeto.

Este evento suele utilizarse en combinación con la propiedad DragIcon del control arrastrado, indicando si dicho control puede o no ser soltado en la posición actual. Para indicar que el control no puede ser soltado en la posición actual, suele utilizarse un icono en forma de señal de prohibición.

```
Private Sub Picture1_DragOver (Source As Control, X As Single,
 Y As Single, State As Integer)
 Select Case State
   Case 0        'Señal de prohibido
Source.DragIcon =LoadPicture
("ICONS\TRAFFIC\TRFFC13.ICO")
   Case 1        'Icono por defecto
Source.DragIcon = LoadPicture()
   Case 2        'Sin actuación
 End Case
End Sub
```

El cuadro 2 contiene un sencillo ejemplo que ilustra el funcionamiento de las operaciones de arrastre. Se trata de un visor de ficheros gráficos (iconos, mapas de bits y metaficheros), en el que debe arrastrarse el nombre del fichero a un cuadro de imagen para que éste sea visualizado. Su aspecto se muestra en la figura 1.

Todos los controles que pueden ser arrastrados disponen de la propiedad DragMode. Se trata de una propiedad

Los eventos MouseDown y MouseUp permiten realizar el control de las pulsaciones del ratón de forma mucho más exhaustiva

CUADRO 2.

1. Añadir un control de lista de unidades (DriveListBox), uno de lista de directorios (DirListBox), uno de lista de ficheros (FileListBox) y un cuadro de imagen (Image) al formulario por defecto, situándolos como se muestra en la figura 1.

2. Establecer en tiempo de diseño las siguientes propiedades:

Form1	Name	frmDragDrop
	Caption	Arrastrar y colocar
Drive1	DragIcon	ICONS\DRAGDROP\DRAG1PG.ICO
Dir1	DragIcon	ICONS\DRAGDROP\DROP1PG.ICO
File1	Pattern	*.bmp;*.ico;*.wmf
Image1	BorderStyle	1 - Fixed Single
	Stretch	True

3. Escribir los siguientes procedimientos de evento:

```
Private Sub Dir1_Change()
  File1.Path = Dir1.Path
End Sub

Private Sub Drive1_Change()
  Dir1.Path = Drive1.Drive
End Sub

Private Sub File1_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As
Single)
  íIniciar arrastre
  File1.DragIcon = Drive1.DragIcon
  File1.Drag
End Sub

Private Sub Image1_DragDrop(Source As Control, X As Single, Y As Single)
  íAñadir "\" si el archivo arrastrado está en el raíz
  If Right(file1.Path, 1) = "\" Then
    dropfile = file1.Path & file1.filename
  Else
    dropfile = file1.Path & "\" & file1.filename
  End If
  íCargar la imagen
  Image1.Picture = LoadPicture(dropfile)
End Sub

Private Sub Image1_DragOver(Source As Control, X As Single, Y As Single, State As
Integer)
  íSi el icono está fuera del área de destino
  If State = 1 Then
    file1.DragIcon = Drive1.DragIcon
  Else
    file1.DragIcon = Dir1.DragIcon
  End If
End Sub
```

4. Salvar el formulario en el archivo DRAGDROP.FRM y el proyecto en el archivo DRAGDROP.VBP.

Ejemplo ilustrativo de operaciones "arrastrar y colocar".

que permite establecer un modo especial de arrastre automático. Cuando esta propiedad se encuentra a 1 (vbAutomatic), cada vez que se ejecuta el evento MouseDown el control entra automáticamente en modo de arrastre, es decir, comienza a ser arrastrado. Por el contrario, cuando esta propiedad se encuentra a 0 (vbManual), el control no entra en modo de arrastre hasta que se ejecute el método Drag explícitamente.

El modo de arrastre automático permite controlar las operaciones de arrastre sin necesidad de activarlas y desactivarlas en el código. Sin embargo, debe tenerse en cuenta que cuando un control tiene puesta a 1 su propiedad DragMode no responde a los eventos en la forma normal. Los eventos MouseDown, MouseUp, Click y DblClick no llegan a producirse, ya que el control entra antes en modo de arrastre. Por otro lado, el evento MouseMove se produce sólo cuando el control no está siendo arrastrado. Por tanto, los resultados de la operación deberán controlarse siempre a través de los eventos DragDrop y DragOver del objeto destino.

LOS EVENTOS DEL TECLADO

Como ya se mencionó al principio de este artículo, Visual Basic permite realizar un control del teclado basado en un cierto repertorio de eventos que posee cada uno de los objetos. Estos eventos se producen al realizar pulsaciones de teclas, y son detectados por el formulario o control que tiene el foco.

Es importante tener en cuenta que Windows trabaja con caracteres del código ANSI (American National Standard Institute). Así, cada carácter está representado por un código de 8 bits, dando lugar a un repertorio de 256 caracteres posibles. Los códigos comprendidos entre 0 y 127 se corresponden a las letras y símbolos del teclado americano, mientras que los comprendidos entre 128 y 255 se corresponden con las letras del alfabeto internacional y otros símbolos especiales. Sin embargo, Visual Basic permite trabajar con el código ASCII (American Standard Code for Information Interchange), más utilizado dentro del mundo de la programación. En cualquier caso, los primeros 128 caracteres de ambos códigos coinciden.

Cuando el usuario pulsa una tecla en tiempo de ejecución, se produce el evento KeyPress. Este evento es detectado por el control o formulario que tiene el foco, ejecutándose el procedimiento de evento correspondiente. Evidentemente, para que un formulario detecte el evento, será necesario que todos sus controles se encuentren ocultos o deshabilitados. Su sintaxis es la siguiente:

```
Private Sub Objeto_KeyPress ([Index
As Integer], KeyAscii As Integer)
```

El parámetro KeyAscii devuelve un número entero que representa el código ANSI de la tecla que ha sido pulsada. Este parámetro es pasado por referencia, de forma que es posible alterar su valor con el fin de pasar al objeto un carácter distinto. Si el valor es cambiado a 0, la pulsación es cancelada y el objeto no recibe ningún carácter. La tabla 1 muestra las teclas cuya pulsación produce el evento KeyPress.

Una vez finalizada la ejecución del procedimiento de evento KeyPress, el carácter correspondiente al valor de

KeyAscii es pasado al objeto, para que éste pueda procesarlo. Así, por ejemplo, en el caso de un cuadro de texto, el carácter que se visualizará cuando el usuario pulse una tecla vendrá determinado por el valor final de KeyAscii, permitiendo corregir o cancelar el valor suministrado por el usuario. De esta forma, para convertir a mayúsculas la tecla pulsada, bastará con realizar la siguiente operación:

```
KeyAscii = Asc(Upper(Chr(KeyAscii)))
```

Al igual que sucedía con las pulsaciones de los botones del ratón, las pulsaciones del teclado también pueden ser controladas de forma más exhaustiva. Para ello, se utilizan dos eventos que se producen separadamente al pulsar y liberar la tecla. Se trata de los eventos KeyUp y KeyDown, que también son detectados por el objeto que tiene el foco. La sintaxis de los procedimientos de evento es la siguiente:

```
Private Sub Objeto_KeyDown ([Index As Integer], KeyCode As Integer,
 Shift As Integer)

Private Sub Objeto_KeyUp ([Index As Integer], KeyCode As Integer,
 Shift As Integer)
```

El parámetro KeyCode devuelve un número entero que representa el código de la tecla pulsada. Sin embargo, en este caso no se trata del código ANSI del carácter, sino de un código de teclado asignado por Visual Basic a cada una de las teclas físicas. Este código también se denomina código de exploración o ScanCode, y sus valores están representados por constantes intrínsecas en la librería de objetos VB, como vbKeyF10, vbKeyPageUp, etc.

El parámetro Shift devuelve un valor entero que indica el estado de las teclas [Shift], [Control] y [Alt] en el momento de producirse el evento. Su funcionamiento es idéntico al explicado para el parámetro Shift de los procedimientos de evento MouseUp y MouseDown.

A diferencia de estos procedimientos de evento, KeyPress no permite detectar la pulsación de las teclas de función (de [F1] a [F12]), las teclas del cursor y las de modificación ([Shift], [Control] y [Alt]). Sin embargo, KeyUp y KeyDown permiten detectar la pulsación de cualquier combinación de teclas.

Los tres eventos de teclado que han sido explicados se producen de forma coordinada, dependiendo del tipo y la combinación de teclas que han sido pulsadas. Estos factores son los determinantes del orden en que se producirán los eventos.

Cuando se pulsa una de las teclas detectables por KeyPress (una letra, un número, etc), el orden en que se producen los eventos es: KeyDown, KeyPress y KeyUp. Sin embargo, si se trata de una tecla no detectable por KeyPress (una tecla de modificación, una tecla de función, etc), sólo se producirán los eventos KeyDown y KeyUp. En el caso de una tecla de modificación, sólo el evento KeyDown reflejará su estado en el parámetro Shift.

Si se pulsa una combinación de teclas, compuesta por una tecla de modificación y otra detectable por KeyPress, el orden en que se producirán los eventos es: KeyDown, KeyDown, KeyPress, KeyUp y KeyUp. El primer KeyDown y el último KeyUp se refieren a la tecla de modificación, mientras el resto se refiere a la tecla detectable, aunque afectada por el estado de la tecla de modificación.

Cuando se mantiene pulsada una tecla detectable por KeyPress, los eventos KeyDown y KeyPress se van produciendo sucesivamente de forma alternada, hasta que la tecla se libera, produciéndose entonces el evento KeyUp. Por el contrario, cuando se trata de una tecla no detectable por KeyPress, sólo se irá produciendo el evento KeyDown.

Aunque los eventos de teclado sólo son detectados por el control que tiene el foco, es posible hacer que el formulario al que pertenece éste también los detecte. Para ello, se utiliza la propiedad KeyPreview del formulario. Así, cuando esta propiedad se encuentra a True, el

CUADRO 3.

1. Añadir un control de lista de unidades (DriveListBox), uno de lista de directorios (DirListBox), uno de lista de ficheros (FileListBox) y un botón de comando (CommandButton) al formulario por defecto, situándolos como se muestra en la figura 3.

2. Establecer en tiempo de diseño las siguientes propiedades:

Form1	Name	frmSendKeys
Caption	Ejemplo de "SendKeys"	
File1	Pattern	*.bmp
Command1	Name	cmdEditar
Caption	&Editar	
FontBold	True	

3. Escribir los siguientes procedimientos de evento:

```
Private Sub Dir1_Change()
File1.Path = Dir1.Path
End Sub

Private Sub Drive1_Change()
Dir1.Path = Drive1.Drive
End Sub

Private Sub cmdEditar_Click()

íSalir si no hay fichero seleccionado
If File1.filename = "" Then Exit Sub

íA–adir "\" si el archivo está en el raíz
If Right(file1.Path, 1) = "\" Then
Fichero = file1.Path & file1.filename
Else
Fichero = file1.Path & "\" & file1.filename
End If

Paint = Shell("C:\Programas\Accesorios\MSPAINT.EXE",1)
SendKeys "%AA" & Fichero & "~", True

End Sub
```

4. Salvar el formulario en el archivo EDITABMP.FRM y el proyecto en el archivo EDITABMP.VBP.

Ejemplo de utilización de la función SendKeys.

formulario detecta los eventos de teclado antes que el control. Por el contrario, si se encuentra a False, el formulario no detecta los eventos.

Los eventos se producen en el mismo orden explicado, aunque alternan entre los del formulario y los del control. Así, se ejecutarán los procedimientos de evento KeyDown del formulario, KeyDown del control, KeyPress del formulario, KeyPress del control, KeyUp del formulario y KeyUp del control, en este orden.

Las alteraciones que se realicen sobre los parámetros KeyAscii y KeyCode de los procedimientos de evento del formulario afectarán al valor de los correspondientes parámetros en los procedimientos de evento del control. De esta forma, desde el formulario es posible controlar las pulsaciones que recibirán sus controles.

Un ejemplo típico de utilización de la propiedad KeyPreview es cuando la aplicación hace uso de las teclas de función para realizar acciones independientes del control que tiene el foco. En este caso, cuando el formulario detecta la pulsación, debe poner a 0 los parámetros KeyCode y KeyAscii, evitando así que el control detecte los eventos de teclado correspondientes.

Para ilustrar el manejo del teclado puede realizarse un sencillo ejemplo, situando un cuadro de texto en un formulario con su propiedad KeyPreview a True. Cuando se pulse la tecla [PgUp], el texto se convertirá a mayúsculas, mientras que al pulsar [PgDn] pasará a minúsculas. Para ello, bastará añadir al formulario el siguiente procedimiento de evento:

```
Private Sub Form_KeyDown (KeyCode As Integer, Shift As Integer)
   If KeyCode = vbPageUp The Text1.Text = UCase(Text1.Text)
   If KeyCode = vbPageDown The Text1.Text = LCcase(Text1.Text)
End Sub
```

GENERACIÓN DE PULSACIONES POR PROGRAMA

Visual Basic dispone de una sentencia especial que permite enviar secuencias de pulsaciones por programa a la ventana que se encuentre activa, sea o no ésta perteneciente a la aplicación, aunque siempre debe tratarse de una aplicación Windows. Realmente, esta sentencia simula dichas pulsaciones, aunque su efecto es el mismo que si el usuario hubiera realizado dicha secuencia de pulsaciones. Se trata de la sentencia SendKeys, cuya sintaxis es la siguiente:

```
SendKeys Secuencia[, Estado]
```

En el argumento Secuencia debe especificarse un string, indicando las pulsaciones a simular en forma de caracteres. Las pulsaciones de teclas alfanuméricas se representan por el propio carácter. Así, por ejemplo, para simular la pulsación consecutiva de las teclas [A], [1] y [X], debe utilizarse el string "A1X".

Los caracteres signo más (+), tanto por ciento (%), control (^), tilde (~) y paréntesis, tienen un significado especial. Por tanto, para especificar las pulsaciones correspondientes será necesario encerrar estos caracteres entre llaves. Así, por ejemplo, para simular la pulsación consecutiva de las teclas [+], [M] y [%], debe utilizarse el string "{+}M{%}".

Cuando se desea especificar la pulsación de la teclas correspondientes a los caracteres de llave ({}), éstos también deben ser encerrados entre llaves a su vez. Los corchetes ([]) no tienen un significado especial, aunque también es recomendable encerrarlos entre llaves, ya que pueden tenerlo en la aplicación destino.

Para especificar las pulsaciones de teclas correspondientes a caracteres no imprimibles ([Intro], [F1], [Tab], etc), se utilizan unos códigos especiales. Estos códigos están compuestos por el nombre de la tecla en mayúsculas y encerrado entre llaves ({ENTER}, {F1}, {TAB}, etc). La tecla [Intro] también puede representarse con el carácter tilde (~). Estos códigos pueden consultarse en la documentación que acompaña a Visual Basic o en su ayuda interactiva.

La pulsación de combinaciones de teclas con las teclas de modificación [Shift], [Control] y [Alt] se simula mediante los caracteres +, ^ y %, respectivamente. Para ello, debe especificarse primero el carácter correspondiente a la tecla de modificación, seguida de la otra tecla. Así, por ejemplo, para especificar la pulsación de las teclas [Ctrl+A] debe utilizarse el string "^A". Si la tecla de modificación afecta a más de una tecla normal, éstas deben ser encerradas entre paréntesis. Así, por ejemplo, para especificar la pulsación de las teclas [Ctrl+A] y [Crtl+C] consecutivas, puede utilizarse el string "^(AC)".

Finalmente, también es posible especificar la pulsación consecutiva de una misma tecla un determinado número de veces. Para ello, debe indicarse el carácter y el número de pulsaciones, separados entre sí por un espacio y encerrados entre llaves. Así, por ejemplo, para especificar la pulsación de la tecla [A] diez veces consecutivas, puede utilizarse el string "{A 10}".

El argumento Estado de SendKeys permite especificar si la aplicación debe esperar a que las teclas enviadas sean procesadas o no. Un valor True indica que la aplicación debe esperar a que se procesen las pulsaciones. Un valor False indica que la aplicación continuará su ejecución normalmente una vez enviadas las pulsaciones.

El cuadro 3 contiene un ejemplo que ilustra el funcionamiento de esta sentencia, permitiendo ejecutar el programa Microsoft Paint desde la aplicación, sin necesidad de intervención por parte del usuario y editando el fichero de dibujo seleccionado.

CONCLUSIÓN

La correcta utilización de los distintos métodos y eventos relacionados con el manejo del ratón y el teclado permiten al usuario de la aplicación realizar las tareas de una forma bastante sencilla e intuitiva. Como siempre, es conveniente que el programador practique con todos ellos, creando sus propios ejemplos. En el próximo artículo se analizará uno de los aspectos mas importantes de Visual Basic, como es la creación y el manejo de gráficos.

CÓMO INCLUIR OBJETOS TRIDIMENSIONALES

Pedro Antón Alonso

En este artículo se verá cómo incluir en nuestros programas las coordenadas y características que definen un objeto tridimensional. Se estudiará el formato 3DS.

Es por todos conocida la existencia de programas de diseño tridimensional, como el LightWave o el 3D Studio. Con estos programas se realizan animaciones profesionales. Evidentemente no estamos hablando de software económico, pero sí de un software muy extendido. No es difícil encontrar en los CD ROM que incluyen las revistas de shareware o freeware mallas diseñadas con el 3D Studio.

EL FORMATO ASC

Una de las formas de almacenamiento del 3D Studio es el formato ASCII. Si se almacena un fichero usando este formato, es posible editar con cualquier editor ASCII la definición del objeto, de esta manera se puede estudiar la forma de almacenamiento empleada por el programa para crear una aplicación auxiliar que nos permita integrar dicha malla en nuestro programa.

Surgen varios problemas como la gran cantidad de espacio usado para este tipo de almacenamiento. Almacenar un número en punto flotante en ASCII nos llevará un byte por cada signo, punto y número; por no hablar del resto de caracteres auxiliares como los espacios, retornos de línea o nombre que identifica cada valor numérico.

Otro problema que surge del empleo de este formato es la conversión de cantidades. Convertir una cadena de caracteres a un valor numérico implica buscar los espacios de la cadena hasta encontrar dónde comienza y dónde acaba el dato a convertir. A lo que se debe añadir el cambio de formato, de ASCII a entero o flotante, dependiendo del tipo de dato a almacenar.

Pero quizás el mayor problema de este tipo de almacenamiento reside en la aparición de palabras clave no controladas por nuestro programa de conversión que pueden cambiar el orden del formato del fichero ASC.

EL PROGRAMA ASC2LIB.PAS

Acompañando a este artículo se adjunta el código de un programa que creará una librería donde irá almacenada la definición de los vértices y las caras del objeto a tratar. Este programa creará una librería con las siguientes características:

Uno de los parámetros de entrada será el nombre de la unidad.

La unidad creará una serie de variables que definirán el objeto a tratar, estas variables podrán ser englobadas en un futuro en una posible definición de una estructura objeto. Las variables en cuestión, son:

NumPointsNOMBRE_DE_LA_LIBRERIA : dword;
NumFacesNOMBRE_DE_LA_LIBRERIA : dword;
NOMBRE_DE_LA_LIBRERIAPoints : PPoints;
NOMBRE_DE_LA_LIBRERIAFaces : PFaces;

A su vez se definen dos procedimientos:

PROCEDURE MakeNOMBRE_DE_LA_LIBRERIA;
PROCEDURE KillNOMBRE_DE_LA_LIBRERIA;

Este programa tiene muchas limitaciones, por ejemplo, sólo funciona con la definición de un objeto y dará problemas con cualquier objeto medianamente complicado. El lector pensará entonces cuál puede ser la utilidad de esta aplicación. Pues bien, resultará, evidentemente, más cómoda esta forma de almacenamiento para una intro o para mallas con el nombre del grupo que sean empleadas más de una vez.

Veamos cómo ha sido creado el programa. De esta forma será mas sencillo para el lector ampliar el programa y adaptarlo a vuestras propias necesidades. Una vez más, desde estas líneas, animar al lector a investigar en el código que acompaña a este artículo, modificándolo a vuestro antojo.

En primer lugar se preparan los parámetros de entrada para conseguir el camino, nombre y extensión adecuadas, forzando la extensión del nombre de la unidad a PAS. Esto permitirá que el programa funcione correctamente tanto si le damos la extensión, como si no.

Se han creado una serie de funciones que faciliten la búsqueda de las palabras o datos deseados en un fichero de texto, estas funciones son:

LookForWord, LookForSpaces y LookForNumber.

- LookForWord busca una palabra en una cadena de caracteres.
- LookForSpaces retorna la posición dentro de una cadena de caracteres donde no hay espacios.
- LookForNumber busca un número en una cadena de caracteres y elimina la parte decimal.

Estas funciones son empleadas para buscar las palabras clave dentro de cada cadena de caracteres leída, separando los lados leídos no necesarios de aquellos que realmente interesan a nuestro programa.

Las siguientes funciones del programa hacen uso de las anteriormente

LISTADO 1.

```
UNIT Ico;

INTERFACE
Uses Lib3DFP;
Var
  NumPointsIco  : dword;
  NumFacesIco   : dword;
  IcoPoints     : PPoints;
  IcoFaces      : PFaces;

PROCEDURE MakeIco;
PROCEDURE KillIco;

IMPLEMENTATION

PROCEDURE MakeIco;
begin
  NumPointsIco:=11;
  getmem (IcoPoints,(NumPointsIco+1)*sizeof(TPoint3D));
  Points^[ 0].x:=-34.0;Points^[ 0].y:= 28.0;Points^[ 0].z:= 46.0;
  Points^[ 1].x:= 34.0;Points^[ 1].y:= 28.0;Points^[ 1].z:= 46.0;
  Points^[ 2].x:= 54.0;Points^[ 2].y:= 28.0;Points^[ 2].z:=-16.0;
  Points^[ 3].x:=  0.0;Points^[ 3].y:= 28.0;Points^[ 3].z:=-56.0;
  Points^[ 4].x:=-54.0;Points^[ 4].y:= 28.0;Points^[ 4].z:=-16.0;
  Points^[ 5].x:=  0.0;Points^[ 5].y:= 62.0;Points^[ 5].z:=  0.0;
  Points^[ 6].x:=  0.0;Points^[ 6].y:=-62.0;Points^[ 6].z:=  0.0;
  Points^[ 7].x:=  0.0;Points^[ 7].y:=-28.0;Points^[ 7].z:= 56.0;
  Points^[ 8].x:= 54.0;Points^[ 8].y:=-28.0;Points^[ 8].z:= 18.0;
  Points^[ 9].x:= 34.0;Points^[ 9].y:=-28.0;Points^[ 9].z:=-44.0;
  Points^[10].x:=-34.0;Points^[10].y:=-28.0;Points^[10].z:=-44.0;
  Points^[11].x:=-54.0;Points^[11].y:=-28.0;Points^[11].z:= 18.0;

  NumFacesIco:=19;
  getmem (IcoFaces,(NumFacesIco+1)*sizeof(TFace3D));
  Faces^[ 0].a:= 0;Faces^[ 0].b:= 5;Faces^[ 0].c:= 1;Faces^[ 0].z:=0;
  Faces^[ 1].a:= 1;Faces^[ 1].b:= 5;Faces^[ 1].c:= 2;Faces^[ 1].z:=0;
  Faces^[ 2].a:= 4;Faces^[ 2].b:= 5;Faces^[ 2].c:= 0;Faces^[ 2].z:=0;
  Faces^[ 3].a:= 4;Faces^[ 3].b:= 0;Faces^[ 3].c:=11;Faces^[ 3].z:=0;
  Faces^[ 4].a:= 0;Faces^[ 4].b:= 7;Faces^[ 4].c:=11;Faces^[ 4].z:=0;
  Faces^[ 5].a:= 0;Faces^[ 5].b:= 1;Faces^[ 5].c:= 7;Faces^[ 5].z:=0;
  Faces^[ 6].a:= 1;Faces^[ 6].b:= 8;Faces^[ 6].c:= 7;Faces^[ 6].z:=0;
  Faces^[ 7].a:= 1;Faces^[ 7].b:= 2;Faces^[ 7].c:= 8;Faces^[ 7].z:=0;
  Faces^[ 8].a:=11;Faces^[ 8].b:= 7;Faces^[ 8].c:= 6;Faces^[ 8].z:=0;
  Faces^[ 9].a:= 7;Faces^[ 9].b:= 8;Faces^[ 9].c:= 6;Faces^[ 9].z:=0;
  Faces^[10].a:= 3;Faces^[10].b:= 2;Faces^[10].c:= 5;Faces^[10].z:=0;
  Faces^[11].a:= 4;Faces^[11].b:= 3;Faces^[11].c:= 5;Faces^[11].z:=0;
  Faces^[12].a:= 4;Faces^[12].b:=11;Faces^[12].c:=10;Faces^[12].z:=0;
  Faces^[13].a:= 4;Faces^[13].b:=10;Faces^[13].c:= 3;Faces^[13].z:=0;
  Faces^[14].a:= 3;Faces^[14].b:=10;Faces^[14].c:= 9;Faces^[14].z:=0;
  Faces^[15].a:= 3;Faces^[15].b:= 9;Faces^[15].c:= 2;Faces^[15].z:=0;
  Faces^[16].a:= 2;Faces^[16].b:= 9;Faces^[16].c:= 8;Faces^[16].z:=0;
  Faces^[17].a:=10;Faces^[17].b:=11;Faces^[17].c:= 6;Faces^[17].z:=0;
  Faces^[18].a:= 9;Faces^[18].b:=10;Faces^[18].c:= 6;Faces^[18].z:=0;
  Faces^[19].a:= 8;Faces^[19].b:= 9;Faces^[19].c:= 6;Faces^[19].z:=0;
end;

PROCEDURE KillIco;
begin
  freemem (IcoPoints,(NumPointsIco+1)*sizeof(TPoint3D));
  freemem (IcoFaces,(NumFacesIco+1)*sizeof(TFace3D));
end;
END.
```

expuestas, para buscar y almacenar los datos buscados.

La función LookForPointsFaces se encarga de buscar el número de caras y de puntos de la malla. Posteriormente dichos valores serán almacenados. Un dato a resaltar, es que el número de puntos varía siempre entre 0 y Numero_De_Puntos-1. Las caras en un fichero de tipo ASC se encuentran definidas tomando el primer vértice como el vértice 0.

La función LookForPoint busca y escribe en el fichero destino las coordenadas de un vértice dentro de una cadena de caracteres.

La función LookForFace busca y escribe en el fichero destino la definición de una cara dentro de una cadena de caracteres. Recordemos que la definición de una cara indica qué vértices conforman dicha cara.

El cuerpo del programa es el encargado de escribir todas las cabeceras y palabras clave del pascal, así como de llamar convenientemente a todas las funciones expuestas creando el fichero deseado, cuyo nombre es el dado como segundo parámetro.

En el Listado 1 se puede observar un sencillo ejemplo de cómo quedaría una librería llamada Ico, con la definición de los vértices y caras de un objeto sencillo, un icosaedro.

EL FORMATO 3DS.

Sobre el formato de este tipo de ficheros no hay ninguna clase de documentación oficial, al menos que yo sepa. Así pues, la información obtenida para escribir esta función, es producto de la investigación de varias personas, así como de las mías propias. Estas personas han puesto el resultado de sus investigaciones a disposición del público en general, usando la conocida red de redes, Internet. Desde estas líneas me gustaría dar las gracias a Jeff Lewis, Javier Arévalo y Bjarke Viksoe.

Un fichero con formato 3DS, está dividido en fragmentos de datos, que denominaremos CHUNKS. Estos fragmentos comienzan siempre de la misma forma, los dos primeros bytes indican qué tipo de información hay en ese fragmento, y los cuatro siguientes la longitud de la información que se va a leer. De esta forma el programa que trate la lectura de este tipo de ficheros, se puede saltar los chunks que desconozca o no interese tratar.

Acompañando a este artículo se incluye el resultado de las investigaciones de Jeff Lewis, donde se pueden observar la práctica totalidad de los chunks que tiene el formato 3DS.

Un fichero con formato 3DS, está dividido en fragmentos de datos denominados chunks

La unidad LIB3DFP que vamos ampliando a lo largo de este curso, crece con este artículo añadiendo una función que permitirá cargar ficheros con formato 3DS, llamada Load3DSObject.

```
FUNCTION Load3DSobject
(filename      : string;
divfactor      : single;
face           : PFaces;
coords         : PPoints;
LookForAmounts : boolean;
VAR NumFaces,
NumVertex  : dword) : boolean;
```

Veamos cómo usar esta función. Resulta absolutamente necesario saber la cantidad de memoria que se va a emplear para cargar la definición de la malla en cuestión. Para ello, en primer lugar se recorre la malla con la variable LookForAmounts = TRUE; el número de caras y el número de vértices de la malla es devuelto en las variables NumFaces y NumVertex respectivamente.

A continuación se reserva la memoria necesaria para almacenar la definición de la malla, de igual forma que lo hace la función InitOBJECT del programa comentado anteriormente, ASC2LIB.

Finalmente se vuelve a llamar a la función Load3DSObject, pero ahora con la variable LookForAmounts = FALSE ya que ahora los punteros face y coords disponen de la memoria necesaria para almacenar la definición de los puntos con los que vamos a trabajar.

Sólo una cosa más a tener en cuenta, en este preciso momento del programa se deben decrementar las variables NumFaces y NumVertex en uno ya que todas las rutinas del programa, así como la definición de las caras del formato 3DS, comienzan con el dato número cero.

Por lo tanto, la función comienza comprobando si existe o no extensión, para ponérsela, en caso de que la cadena de caracteres dada como nombre no la lleve. Acto seguido se lee un chunk que debe ser 4D4Dh, este valor identifica a un fichero del programa 3DS.

Una vez verificado que el fichero tiene el formato 3DS se leen los chunks que vengan uno detrás de otro hasta llegar a los que son de nuestro interés, a saber:

- 3D3Dh Identifica una definición de malla. El programa genera más tipos de fichero con distinta extensión, como por ejemplo PRJ, MLI, o MAT, identificados con otro chunk de cabecera distinto.
- 4xxxh Los chunks pertenecientes a este grupo contienen la definición de una malla y sus componentes.
- 4000h Describe el nombre del objeto. Indicando el nombre de éste con una cadena, como ya se ha dicho, que acaba con el carácter cero. Puede ser usado para usar sólo algunos de los objetos que contenga el fichero, desechando el resto.

- 4100h Este chunk va siempre seguido de una matriz de vértices, una matriz de banderas de vértices, una matriz de transformación del objeto y una matriz conteniendo la definición de las caras. Pero únicamente indica el comienzo de estos datos, no hay datos que leer, excepto el resto de la definición.

- 4110h Describe los vértices de un objeto. En primer lugar hay dos bytes que indican su número, seguido de su definición. Los puntos constan de tres valores (x,y,z), valores de tipo flotante y de tamaño 4 bytes. El tipo punto sería entonces:

```
TPunto3DS = record
          x,y,z : single;
        end;
```

- 4120h Describe las caras de un objeto. Los dos primeros bytes indican el número de caras, y a continuación la definición de las caras. Las caras constan de tres valores (A,B,C) de dos bytes, y de una bandera, también de dos bytes. El tipo cara queda entonces:

```
TCara3DS = record
         a,b,c : word;
         flag  : word;
       end;
```

- 4130h Describe el material empleado por las caras de un objeto. En primer lugar, se almacena el nombre de dicho objeto en una cadena terminada con el carácter cero. A continuación, el número de caras que usan ese material y cuáles son. Tanto el número de caras como las caras que lo usan son palabras de dos bytes.

- 4600h Describe la definición de una luz omnidireccional. Sus coordenadas son idénticas a las de un punto, pero van seguidas de un chunk de color. De esta forma, el tipo luz queda:

```
TLuz3DS = record
        x,y,z : single;
      end;
```

- 4700h Describe la localización de una cámara. Ésta viene dada por varios valores. En primer lugar, como si de un punto se tratara, se indica dónde va emplazada la cámara, es decir las coordenadas (x,y,z) donde se encuentra (lógicamente, todas las coordenadas son de tipo flotante y de tamaño cuatro bytes, como ocurre con los puntos). A continuación, se almacena el punto hacia donde está mirando la cámara. También almacena en otro flotante el ángulo que está girada la cámara. Y finaliza con el foco de la lente que emplea la cámara. El tipo cámara queda entonces:

```
TCamara3DS = record
           x0,y0,z0 : single;
           xf,yf,zf : single;
           Angulo   : single;
           Foco     : single;
         end;
```

- AFFFh Indica la entrada de un material. Aunque realmente la definición completa de un material corresponde a todo el grupo Axxxh, este chunk puede ser usado para leer el nombre de un material.

- B000h Describe datos del keyframer. El keyframer es un módulo del 3D Studio empleado para generar animaciones. El código que acompaña a este artículo únicamente se usa para indicar el fin de la lectura, es decir, no se leerán mas datos del fichero si se cumplen las definiciones dadas por este módulo.

El keyframer es un módulo del 3D Studio para generar animaciones

Vistos los chunks tratados pasemos a describir el funcionamiento de la función incluida en el código que acompaña a este artículo:

La función posee una constante global _DEBUG_3DS que nos permite ver por pantalla si los datos leídos son correctos. Una vez leído y detectado el chunk 3D3Dh se procede a buscar el siguiente que nos interesa, el 4000h que contiene la definición de un objeto, si la constante _DEBUG_3DS se encuentra activada, una vez encontrado este identificador veremos en la pantalla el nombre de este objeto.

Los datos que se deben leer a continuación serán el número de vértices y caras, si es la primera vez que llamamos a la función, o bien la definición de éstas si es la segunda vez que lo hacemos. Se buscará entonces el identificador 4000h que nos dará el comienzo de los auténticos datos.

Encontrado ese chunk procedemos a buscar los siguientes de su grupo que son los que contienen la información realmente interesante para nosotros. Principalmente 4110h y 4120h, que son los que almacenan la información de las caras y los vértices que definen el objeto encontrado. Además se ha dejado el 4130h que describe el material empleado por las caras del objeto, para posibles ampliaciones del código, bien sea por el lector o bien en siguientes artículos.

Una vez concluida la lectura de los datos que definen un objeto, el programa continúa leyendo los datos de las cámaras o luces hasta que se encuentre con un chunk perteneciente a otro grupo, dicho sea de otro modo, si no pertenece al grupo 4xxxh. En cuyo caso el programa sale en busca de otro objeto, (chunk 4000h) o de la definición de un material. Saldrá en busca de otra definición de objeto si encuentra la definición de los datos del módulo keyframer.

La rutina finaliza cuando termina de leer el fichero o, lo que es lo mismo, cuando se produce un error de entrada / salida, indicado por la función IOResult.



PROGRAMACIÓN CON EL LENGUAJE TCL (II)

Carlos Gerardo Pérez Pérez



En el número de Diciembre se presentó la potencialidad que se puede obtener al emplear en conjunto el lenguaje TCL a su toolkit de desarrollo gráfico, conocido como TK. En éste se describirán los elementos necesarios para programar utilizando el lenguaje TCL; el presente artículo trata sobre los fundamentos del lenguaje, sus reglas básicas de sintaxis, los mecanismos utilizados por el intérprete de TCL y los principales comandos del lenguaje.

El TCL es un lenguaje de comandos basado en caracteres, es decir, que es interpretado, lo que implica la existencia de un "*parser*" que interpreta las palabras (los comandos) y genera el resultado(ver figura 1).

El lenguaje tiene pocas construcciones fundamentales y éstas de una sintaxis relativamente reducida, lo cual lo hace fácil de aprender. Los mecanismos básicos están todos relacionados a caracteres y a la sustitución de caracteres, por lo que no es complicado visualizar qué es lo que está ocurriendo en el intérprete.

Los mecanismos de sintaxis y reglas de sustitución en el lenguaje TCL son muy simples aunque en un principio puedan parecer complicados

Para trabajar con el TCL se emplean los intérpretes de comandos *tclsh* o *wish*. El primero es una *shell* simple de TCL y el *wish* es un intérprete que además de proporcionar las capacidades de *tclsh* añade la posibilidad de crear interfaces gráficas; para ello, al momento de ejecutarlo crea una ventana en blanco que se transformará dependiendo de los comandos que sean utilizados.

SUSTITUCIÓN DE VARIABLES

Como en cualquier lenguaje de programación se pueden utilizar variables para guardar información mientras dura la ejecución de algún programa. Como ya se mencionó, el TCL interpreta toda la información como una secuencia de caracteres; ese es el motivo de que no sea necesario hacer ninguna diferenciación entre las variables que guardarán valores numéricos y las que sólo guardarán caracteres.

Sin embargo, a diferencia de muchos lenguajes que son estructurados, en TCL no es necesario declarar las variables que se utilizarán: únicamente se incluyen en el código del programa y serán asignadas en tiempo de ejecución. El comando *set* crea una variable y le asigna un valor; su sintaxis es la siguiente:

set nom_var valor

donde *nom_var* es el nombre de la variable en la que se almacenará "valor".

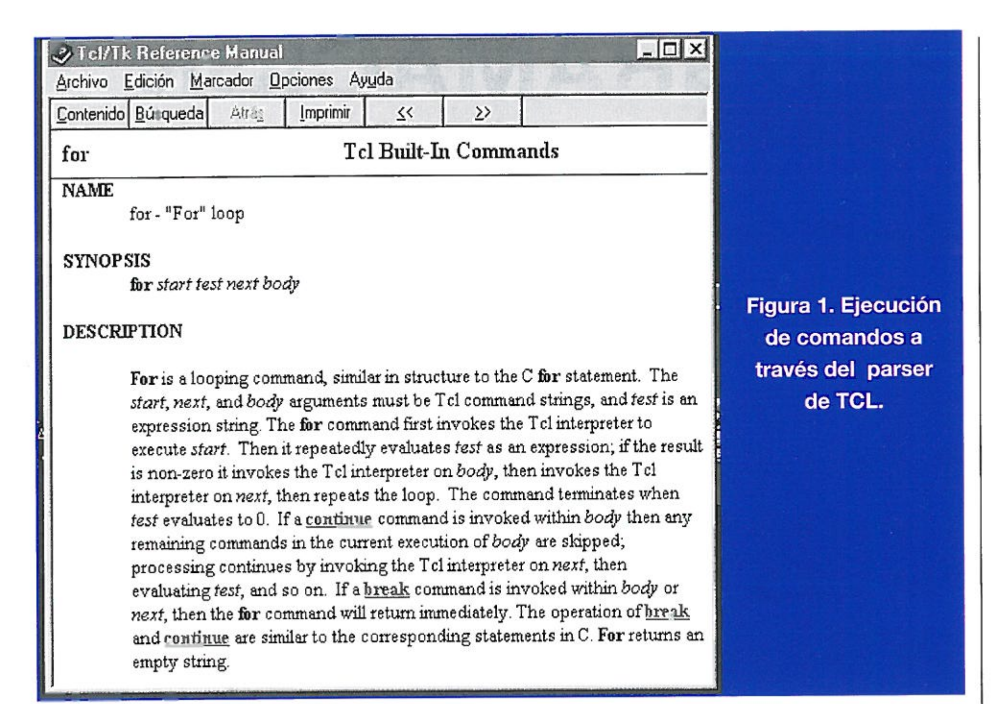
Por ejemplo:

set valor 28

Cuando se ejecute este comando aparecerá un 28, que es el valor asignado a la variable llamada "valor".

Si quisiéramos asignar ahora el valor de la variable "valor" a la variable "nuevo" se podría pensar en hacer lo siguiente:

set nuevo valor


Tcl/Tk Reference Manual

Archivo Edición Marcador Opciones Ayuda

Contenido Búsqueda Atrás Imprimir << >>

for **Tcl Built-In Commands**

NAME

for - "For" loop

SYNOPSIS

for *start test next body*

DESCRIPTION

For is a looping command, similar in structure to the C **for** statement. The *start*, *next*, and *body* arguments must be Tcl command strings, and *test* is an expression string. The **for** command first invokes the Tcl interpreter to execute *start*. Then it repeatedly evaluates *test* as an expression; if the result is non-zero it invokes the Tcl interpreter on *body*, then invokes the Tcl interpreter on *next*, then repeats the loop. The command terminates when *test* evaluates to 0. If a continue command is invoked within *body* then any remaining commands in the current execution of *body* are skipped; processing continues by invoking the Tcl interpreter on *next*, then evaluating *test*, and so on. If a break command is invoked within *body* or *next*, then the **for** command will return immediately. The operation of break and continue are similar to the corresponding statements in C. **For** returns an empty string.

Figura 1. Ejecución de comandos a través del parser de TCL.

Pero el ejemplo anterior asigna el valor "valor" como cadena a la variable "nuevo", y no el número 28 como se deseaba.

Para asignar correctamente el contenido de la variable es necesario indicarlo anteponiendo el símbolo $ al nombre de la variable; es decir:

set nuevo $valor

De esta forma la variable "nuevo" contendrá el valor que contiene "valor", y no su nombre como en el caso anterior. Lo que hace el carácter $ es una sustitución del nombre de la variable por su contenido.

SUSTITUCIÓN DE COMANDOS

La utilización del símbolo $variable es uno de los casos de sustitución permitidos por el TCL. Otro caso es la sustitución de comandos. Es posible asignar a las variables resultados de operaciones anidando los comandos necesarios para realizar dichas operaciones.

Para indicar un comando anidado sólo es necesario encerrarlo entre corchetes [], así el intérprete toma todo lo que se encuentra entre éstos y lo evalúa como un comando. Por ejemplo:

set suma [expr 17+25]

Si esto se escribiera sin los corchetes se generaría un error.

En este caso la expresión 17+25 es evaluada y su resultado se deposita en la variable suma; el *shell* muestra el resultado de la última operación, en este caso 42.

Es posible utilizar anidamientos sucesivos, por ejemplo:

set num [expr [string length Palabra] / 2.0]

El comando *string* tiene varios usos: en este ejemplo se le pidió que devolverá la longitud de la cadena (*length*) que se le dio como tercer parámetro (Palabra) a la instrucción; en este caso mide 7 caracteres de largo; este resultado lo dividió entre 2 y lo guardó en la variable prueba. La instrucción *string* será analizada con mayor detalle en el siguiente capítulo.

En el ejemplo anterior se realizó la división entre 2.0, debido a que el TCL devuelve por defecto valores enteros siempre y cuando se le hayan dado datos enteros; para que devuelva valores con decimales o reales es necesario que cuando menos uno de los valores sea real:

expr [expr 3/2]+1

Salida: 2

expr [expr 3.0/2]+1

Salida: 2.5.

El número de dígitos que muestra el TCL esta predefinido en 6, pero es posible cambiarlo modificando para ello el valor de la variable *tcl_precisión* a la cantidad de dígitos que se deseen:

expr 10/3

salida: 3

expr 10/3.0

salida 3.33333

set tcl_precision 12
expr 10/3.0

Salida: 3.33333333333

ESTRUCTURAS DE CONTROL

Para lograr darle al programa un cierto grado de "lógica" es necesario indicarle acciones a realizar dependiendo de determinados eventos; por ejemplo, se puede necesitar un programa que muestre el resultado de una operación en la pantalla siempre y cuando ese resultado sea igual a cierto patrón; esto se lograría con una condición sencilla, que sería: si a es igual a b entonces muéstralo en pantalla; en otro caso muestra un mensaje.

TCL es un lenguaje de comandos integrable en programas en C

IF THEN ELSE

En TCL la condición sencilla se logra por medio de la estructura *"if condicion then accion1 else accion2"*; la condición debe ser *booleana*, es decir, que se pueda evaluar como falsa o verdadera; *accion1* es una instrucción o un conjunto de instrucciones que se ejecutarán en caso de que la condición se cumpla; de no cumplirse se ejecutarán las instrucciones incluidas en *accion2*.

Sintaxis:

if {condición} { instrucciones } else { instrucciones }

Por ejemplo:

```
if {$valor == 5} {
 puts stdout "Valor 5, se cumplió la condición"
} else {
     puts stdout $i
     }
 puts stdout "Condición no cumplida"
}
```

El parser de TCL no asigna significado a los argumentos, ya que son los mismos comandos los que les dan algún significado

```
#!/usr/local/bin/tclsh
set valor [gets stdin]
if {$valor == 5} {
 puts stdout "Valor 5, se cumplió la condición"
} else {
     puts stdout $i
     }
 puts stdout "Condición no cumplida"
}
```

SWITCH

En ocasiones es necesario realizar varias comparaciones del contenido de una variable para realizar cierta acción dependiendo de su valor específico.

Para evitar escribir una larga lista de estructuras IF THEN ELSE el TCL cuenta con la estructura SWITCH, que compara una variable con una lista de valores, y dependiendo de cuál sea este valor es la instrucción que se ejecutará.

Sintaxis:

```
switch -exact — $variable {
  caso1 {accion1}
  caso2 {accion2}
  caso3 {accion3}
  default {accion4}
}
```

Ejemplo:

```
#!/usr/local/bin/tclsh
 puts stdout "Dame un numero entre 1 y 3"
 set opcion [gets stdin]
 switch -exact — $opcion {
  1 {puts stdout "Uno"}
  2 {puts stdout "Dos"}
  3 {puts stdout "Tres"}
  default {puts stdout "Ninguna"}
 }
```

foreach

Esta estructura de datos toma uno por uno los valores de una lista y ejecuta para cada uno de ellos las instrucciones indicadas.

Sintaxis:

foreach value { lista } { acción }

Ejemplos:

```
set i 1
foreach value {1 2 3 4 5 6 7 8} {
 set i [expr $i*$value]
}
puts stdout $i
```

```
#!/usr/local/bin/tclsh
 set i 1
foreach value {1 2 3 4 5 6 7 8} {
 set i [expr $i*$value]
 }
puts stdout $i
```

ITERACIONES WHILE

Una iteración WHILE es un conjunto de instrucciones que se repiten mientras la condición que controla el ciclo sea verdadera. Por ejemplo, si queremos hacer que nuestro programa cuente del 1 al 10, necesitamos hacer lo siguiente:

```
set contador 1
while {$contador <= 10} \
 {puts stdout $contador
 incr contador }
```

La sentencia encerrada entre las primeras llaves es la condición que se evalúa; si es verdadera se ejecuta el conjunto de instrucciones que se encuentran entre el segundo par de llaves.

while

Sintaxis

while { ExpresionBooleana } { accion }

Ejemplos:

```
set 1 0
while {$i<10} {incr i}
#!/usr/local/bin/tclsh
```

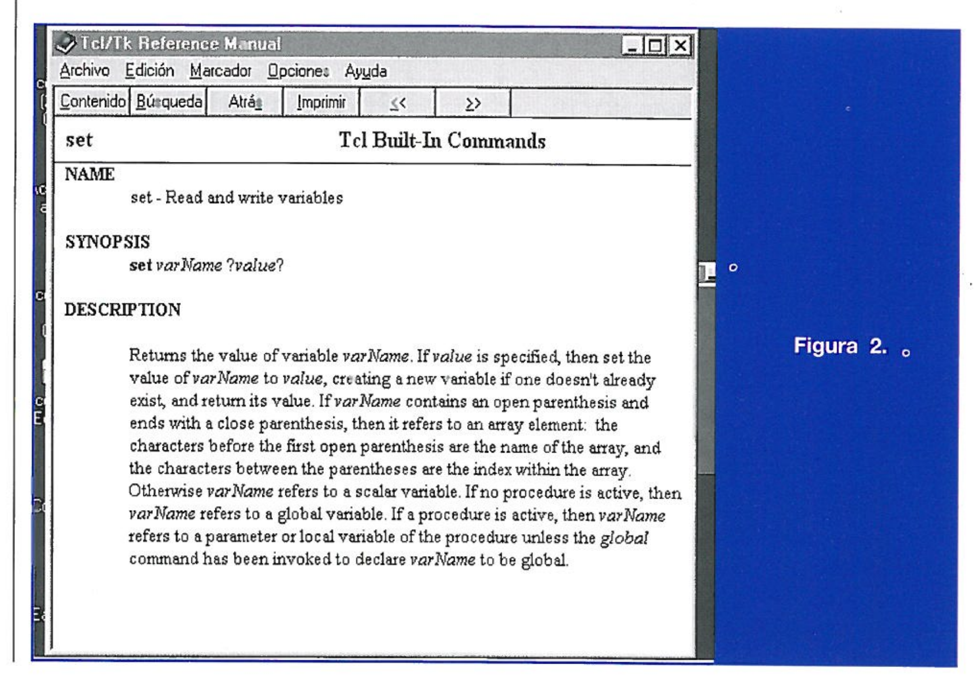


Figura 2.

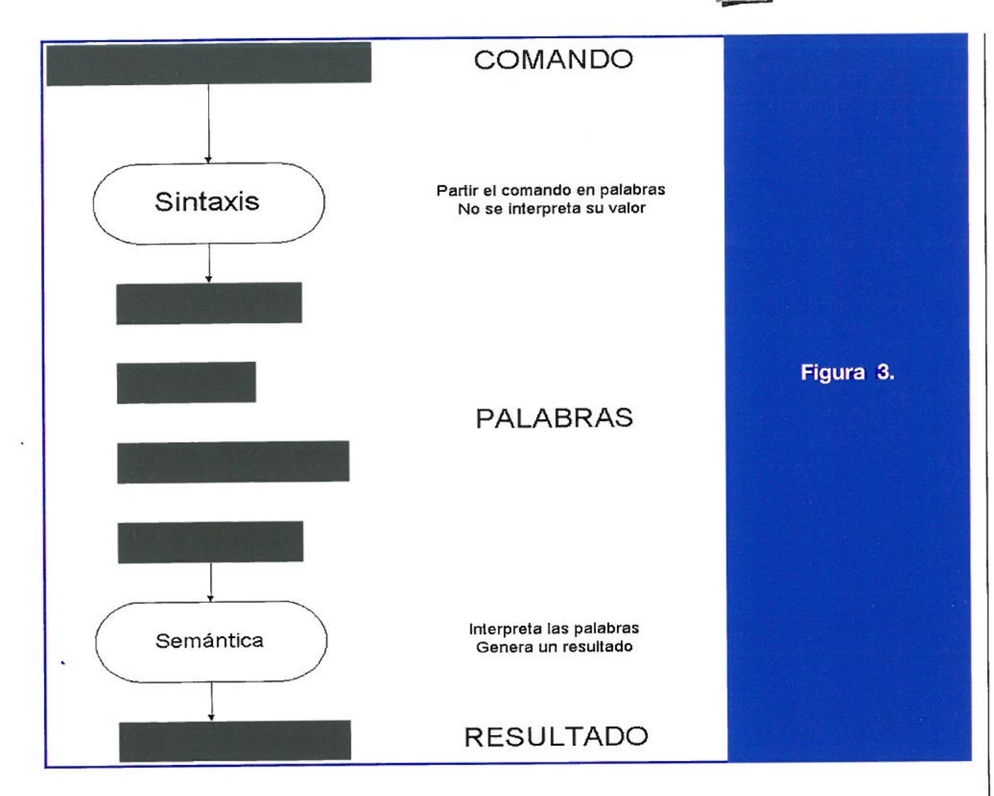


Figura 3.

```
set contador 1
while {$contador <= 10} \
  {puts stdout $contador
  incr contador }
```

FOR

Una instrucción FOR realiza un ciclo de instrucciones un número de veces utilizando un contador que le indica la cantidad de ciclos. Dicho contador se inicializa a un cierto valor, luego se coloca la condición a evaluar y por último la forma del incremento.

Sintaxis:

for { inicializacion } { condicion } { incremento } { accion }

Ejemplo:

```
for {set i 0} {$i < 10} {incr i 3} {
   lappend aList $i
}
puts stdout aList
```

BREAK

Causa la terminación inmediata de un ciclo FOR

CONTINUE

Ocasiona la terminación del ciclo en proceso para continuar con el siguiente.

PROCEDURES

Para la definición de PROCEDURE se emplea el comando *proc*, con el cual se pueden crear nuevos comandos que estarán definidos dentro de la aplicación TCL que se está creando.

proc

Sintaxis:

proc nombre argumentos cuerpo

Ejemplo:

```
proc sub1 x {expr $x-1}
```

Por lo que el PROCEDURE se comporta como un comando predefinido:

sub1 3 devuelve 2

Asimismo, se pueden declarar valores por defecto de los argumentos; por ejemplo:

```
proc decr  {x {y 1}}
{expr $x-$y}
```

en caso de que sólo se declare el valor de x, el valor por defecto de y será 1.

Además, si se quieren tener valores fijos dentro de los PROCEDURES será necesario emplear una variable global.

Por ejemplo:

```
global fecha
set fecha 010297

proc cal a {
global fecha
set hoy $fecha

}
```

RESUMEN

En el presente artículo se han presentado los mecanismos de manipulación de variables, comandos, estructuras de control y *procedures*, por lo que en el próximo artículo de la serie se mostrará la forma de manipulación de las cadenas de caracteres, el acceso a ficheros y la manipulación de errores.

BIBLIOGRAFÍA

John K. Osterhout, *Tcl and the Tk Toolkit*, Addison Wesley, 1994
Brent B. Welch, *Practical Programming in Tcl and Tk*, Prentice Hall, 1995
News: comp.lang.tcl

LENGUAJES Y HERRAMIENTAS (II)

Enrique de Alarcón

Prosigue el análisis de los lenguajes de programación y herramientas de desarrollo más importantes o populares que existen hoy en día para DOS, Windows y otras plataformas.

Hasta la aparición del lenguaje Java, se creaban programas para Web solamente con el interface CGI (Common Gateway Interface).

LENGUAJE JAVA

El Java nació originariamente como un lenguaje pensado para el desarrollo de aplicaciones pequeñas destinadas a PDA's y otros dispositivos electrónicos pequeños dado que el lenguaje C/C++ no les servía totalmente para tal fin dadas sus características generales.

Debido precisamente a estar pensado para la creación de aplicaciones pequeñas, compactas y funcionales, cumplió las condiciones necesarias para que se convirtiera en el candidato perfecto para ser usado como lenguaje dentro de las HTML de Internet para la generación de miniprogramas destinados a la interactuación usuario-Red.

Como características generales, Java es un lenguaje de 3ª generación parecido al C++ (del cual partió su creación) con el que se pueden realizar también aplicaciones *normales* no-internet.

Su principal diferencia frente al C es que incorpora capacidad internet, entendiendo como tal la capacidad que posee para la creación de Applets y la incorporación de los protocolos TCP/IP, HTTP y FTP.

Como se ha dicho, es un lenguaje orientado a objetos y por consiguiente dispone tanto de encapsulación como de herencias, polimorfismos, tipos de clases y otras características típicas. Además de todo esto, al igual que pasaba con el REXX de OS/2, permite la ejecución Multithread.

Otra característica de agradecer es que las aplicaciones Java son muy fiables y es muy improbable que se bloqueen en tiempo de ejecución, debido todo ello a que es un lenguaje más estructurado que el C++ (menor número de fallos posible) y a que es mucho más flexible.

Debido a que las aplicaciones que creemos en Java no son EXE's sino ficheros de pseudocódigo y a que necesitan un run-time para poder ejecutarse, hace que no sea necesaria su recompilación para poder ser usado indistintamente en diferentes plataformas (sólo cambia el Run-time), puesto que es el run-time el que se encarga de adaptar el pseudocódigo de la aplicación a la plataforma en la que corre.

Como curiosidad, en el listado 1 podemos encontrar un ejemplo de un fuente Java donde se ve su parecido con el C++ y otros.

LISTADO 1.

```
/*-----------------------------------------------------------
-*/
/*                                                        */
/* EJEMPLO DE PROGRAMA JAVA   */
/*-----------------------------------------------------------
-*/
import java.util.Date;
class Hora_Actual
  {
   public static viod main
   (string args [ ] ) {
       Date DATO= new Date();
   System.out.println( DATO );
                          }
  }
```

A continuación detallamos los dos paquetes Java que podemos usar para programar en este lenguaje, uno de los cuales acaba de aparecer en el mercado:

Sun JAVA:

Hasta ahora, el único entorno de desarrollo Java que ha existido a disposición de los programadores ha sido el de los propios creadores del lenguaje, que no es otro que Sun. Este entorno se encuentra disponible para Windows 95, Windows NT 3.5 y 3.51, en SPARC y en Solaris 2.3 - 2.5.

MS Visual J++:

El Microsoft Visual Java++ es un entorno de desarrollo visual de Java totalmente orientado a objetos y posee todas las características del lenguaje JAVA original de SUN. MS-Visual J++ es un entorno de desarrollo visual de estilo muy parecido al MS-Visual C++ que se basa en el nuevo lenguaje J++, una versión ampliada del JAVA de SUN. Podríamos decir que J++ es el lenguaje JAVA de SUN con posibilidad de usar controles ActiveX propios de Windows 95.

Pese a las posibilidades que se le han añadido, los fuentes que desarrollemos con él son compatibles 100% con el compilador JAVA de SUN. El entorno de desarrollo, que es muy potente, incluye el programa principal de control general, el compilador J++, un depurador y un sistema completo de ayuda ON-LINE.

Como Herramientas visuales incluye el Java Applet AppWizard, que permite la creación totalmente visual de los famosos Applets. A parte de esto el sistema soporta otras herramientas de la misma Microsoft y de terceras firmas para ayudar en el desarrollo.

Como último detalle, decir que este entorno sólo funciona en equipos potentes y sobre Windows 95 y NT.
Autor: Microsoft Corporation.

LENGUAJE ADA

El lenguaje ADA fue creado por la *U.S.Departement of Defense* (DoD) para su uso en sistemas de tiempo real como son sistemas de seguimiento de misiles, bases de control, radares, torpedos, etc... donde se implementaba de forma habitual directamente en hardware. Una consecuencia directa de su origen, es que es un lenguaje muy rápido y seguro.

La primera versión ADA estándar ANSI apareció en el año 1983 y tras su popularización de estos últimos años, se crearon nuevos estándares renovados del mismo. Los más conocidos son el ADA 9X y el ADA95, los cuales están aún en evolución.

Como características generales del lenguaje tenemos que está totalmente orientado a objetos, posee sistemas de gestión de excepciones para control de errores como desbordamientos y similares, puede ejecutar en *multithread* y posee una extensa librería de funciones, además de poderse enlazar con casi cualquier lenguaje.

En la actualidad existen muchos compiladores ADA como, por ejemplo, el que se dio como regalo en el CD-ROM del Sólo Programadores número 20, donde también se incluía, para los interesados, un completo manual de aprendizaje e información ADA en formato hipertexto de Internet.

En el listado 2, podemos ver un programa de ejemplo del lenguaje ADA.

LISTADO 2.

```
— EJEMPLO PROGRAMA EN ADA          ..
— ........................................
with Ada.Text_IO;
procedure ESCRIBE is
begin
 Ada.Text_IO.Put_Line("Texto de ejemplo");
 Ada.Text_IO.Put_Line("en lenguaje ADA.");
end ESCRIBE;
```

LENGUAJES PARA BASES DE DATOS

Si lo que se necesita es crear programas para la gestión de bases de datos, existen bastantes lenguajes conocidos, algunos de los cuales tienen mucha calidad y años de evolución y experiencia.

A continuación detallamos los principales lenguajes y entornos de desarrollo que podemos encontrar:

DBASE 5.0

Este es un entorno de desarrollo muy completo. Utiliza una versión actualizada del famoso lenguaje de programación Dbase, que tanta fama tiene y está totalmente orientado a objetos. Se caracteriza por la facilidad de aprendizaje y por la gran flexibilidad que tiene a la hora de programarse.

A grandes rasgos podríamos decir que tiene bastante similitud con el lenguaje CA-CLIPPER y dispone de clases para controles WINDOWS y el Browser además de ser compatible OLE, DDE.

VISUAL FOXPRO 3.0. EDICIÓN PROFESIONAL

Es un entorno de desarrollo basado en programación visual. Con él podemos crear aplicaciones tipo Cliente/Servidor tanto de 16 como de 32 bits para Windows95 y Windows NT y dispone de muchas herramientas visuales, con lo que el desarrollo de una aplicación puede hacerse prácticamente sin escribir código de programa. Además de todo ello, posee también controladores ODBC de 32 bits que le dan la posibilidad de crear aplicaciones que pueden enlazar con casi cualquier tipo de servidor.

Es un lenguaje orientado a objetos y eventos y puede utilizar las clases VCX. Posee también la capacidad de soportar objetos OLE OCX 32 bits que son, según muchos analistas, un elemento que tendrá mucha importancia en la programación del futuro próximo.

MULTIBASE 2.0

Es un entorno completo de desarrollo que funciona tanto en DOS como en WINDOWS y UNIX y que no requiere muchos recursos del sistema para su buen funcionamiento (4Mbytes). Las aplicaciones que realicemos con él se caracterizan porque se parecen mucho entre ellas, indistintamente de la plataforma en la que corran, y su estilo es muy parecido al GDI del Windows.

El programa que realicemos con él, cuando funcione en Windows, que será lo más probable, no podrá usar multitarea, pero podrá soportar DDE, funciones del API Windows y los Windows Sockets para Red además de ser compatible OLE.

El Lenguaje en que se basa es el CTL, un lenguaje de 4ª generación basado en el SQL. Es un lenguaje de tipo procedural, bien estructurado y tiene una librería con más de cien funciones. Otro detalle a destacar es que permite enlazar con C mediante el uso de DDL's.

Los programas que genera este lenguaje son rápidos y fiables, funcionan en todas las plataformas sin necesidad de recompilación (muy importante) y tanto en red como en monopuesto sin variaciones.

Autor del entorno: TransTools S.A.

CA-CLIPPER 5.2

El lenguaje CLIPPER, que apareció hace ya once años, es uno de los más usados por los programadores profesionales de aplicaciones relacionadas con el almacenamiento de datos.

El CA-CLIPPER 5.2, permite poder utilizar ficheros Dbase, Dbase IV, Dbase III, Fox Pro y Paradox. Además también está preparado para la compartición de ficheros por varias aplicaciones simultáneas.

El entorno dispone de muchas opciones y bastantes utilidades, para enlace, depurador Clipper, etc...

Autor: Computer Asociates.

LENGUAJE FORTRAN

El lenguaje Fortran nació en 1955 después de que, en 1954, John Backus creara las bases del mismo.

La primera versión estándar ANSI de este lenguaje se aprobó el 3 de Abril de 1978 y se le llamó Fortran 77. La palabra FORTRAN proviene del inglés FORmula TRANSLATOR que se traduce al español como Traductor de fórmulas. Dicho esto, queda claro que es un lenguaje creado para su uso exclusivo en aplicaciones matemáticas de todo tipo.

Este lenguaje debe su fama a que fue el primero que se usó a nivel mundial como lenguaje de programadores profesionales y, gracias a ello, evolucionó mucho durante las primeras décadas de la informática a la vez que se adaptó para muchas plataformas diferentes. Así, en el año 1958, apareció la versión FORTRAN II y, en 1962, la versión FORTRAN IV.

La principal característica de este lenguaje es que su sintaxis se parece mucho a la sintaxis usada por los científicos en la escritura de fórmulas, lo cual se convierte en una gran ayuda para los programadores de aplicaciones matemáticas.

Actualmente existen varios compiladores FORTRAN modernos disponibles en el mercado, aunque el más profesional quizás sea el Microsoft FORTRAN Powerstation, que es un entorno de desarrollo Windows al estilo de otros productos de la misma casa.

En el listado 3 presentamos un pequeño ejemplo de este lenguaje:

LISTADO 3.

```
EJEMPLO DE FORTRAN
______________________________________
—
FUNCTION RAIZ (N1, P, TERMEC)
  REAL N1
  X1= (-P+SQRT(P**2-4.*N1*TERMEC))/(2.*N1)
  X2= (-P-SQRT(P**2-4.*N1*TERMEC))/(2.*N1)
  RAIZ= AMAX1(X1,X2)
  RETURN
END
```

OTRAS HERRAMIENTAS MATEMÁTICAS

Aparte de lenguajes orientados a la creación de aplicaciones matemáticas como el FORTRAN que acabamos de explicar, existen otras herramientas especializadas para ser usadas en aplicaciones matemáticas de alto nivel. Dentro de este grupo, el mayor exponente lo encontramos en el programa MATHEMATICA, una aplicación que va por su reciente tercera versión y que lleva más de seis años en el mercado.

Sus capacidades de cálculo matemático científico cubren las necesidades del profesional más experto que pueda haber y el interfaz con el usuario se ha mejorado mucho respecto a la versión 2, el cual ha pasado de ser una línea de comandos a un interfaz gráfico de menús e iconos típico de cualquier programa Windows actual que además es totalmente personalizable a gusto del usuario (mejor dicho, del matemático). Las áreas en las que puede ser usado son muchas, aunque principalmente se utiliza en ingeniería eléctrica, mecánica, nuclear, aerospacial.... y en ciencias de computación con todas sus variantes.

El lenguaje Mathematica se diferencia de casi todos los demás, permite dos formas de programación diferentes, la procedural y la funcional. En el listado 4 podemos ver ejemplos de ambos tipos de programación en este lenguaje.

Otra característica que destaca mucho en este entorno matemático es que todos los cálculos que realizamos pueden mostrarse en forma de gráficos tridimensionales que poseen una calidad espectacular y que se pueden exportar como bitmaps a otros programas en varios formatos gráficos estándar, como es, por ejemplo, el GIF.

LISTADO 4.

```
1-EJEMPLO PROCEDURAL:
   exprod[n_Integer] :=
            Block[ { cont,i },
               cont=1;
               For [ y=0, i<n, y++,
                  cont= (x+i) cont];
               cont= Expand[cont];
               Return[cont];
            ]
2-EJEMPLO FUNCIONAL:
   exprod[n_Integer] := Expand[ Product[ x+i, { y,
n } }
```

En general, podemos decir que es el programa más veterano y de mayor calidad dentro de este área informática el cual está disponible para muchas plataformas, incluyendo a Macintosh, Intel, Convex, Data General,Hewlet Packard, IBM Risc, MIPS... etc...

Autor: Wolfram Research Inc.

INTELIGENCIA ARTIFICIAL

El número de profesionales que requieren herramientas para la creación de aplicaciones de inteligencia artificial es muy reducido debido a que, en general, su mercado resulta bastante escaso.

Lo que es cierto es que, a pesar de lo dicho, hay varios lenguajes específicos para I.A. y los más conocidos son el famoso LISP y el PROLOG.

LISP son las siglas que provienen del inglés LISt Processing que significa literalmente *procesamiento de listas* y que se basa en la metodología de proceso funcional. Este lenguaje fue creado en Estados Unidos.

El PROLOG, que proviene de las palabras inglesas PROgramming in LOGic significa literalmente *programación de lógica* y es muy parecido al LISP.

Todos estos lenguajes se caracterizan porque están destinados a facilitar el almacenamiento de conocimiento y datos, así como su posterior uso.

También están apareciendo en la actualidad herramientas destinadas a la creación de redes neuronales para realizar aplicaciones basadas en soluciones heurísticas así como programas para someterlas a aprendizaje, aunque su desarrollo está muy poco evolucionado y son limitados los programas disponibles actualmente para el programador.

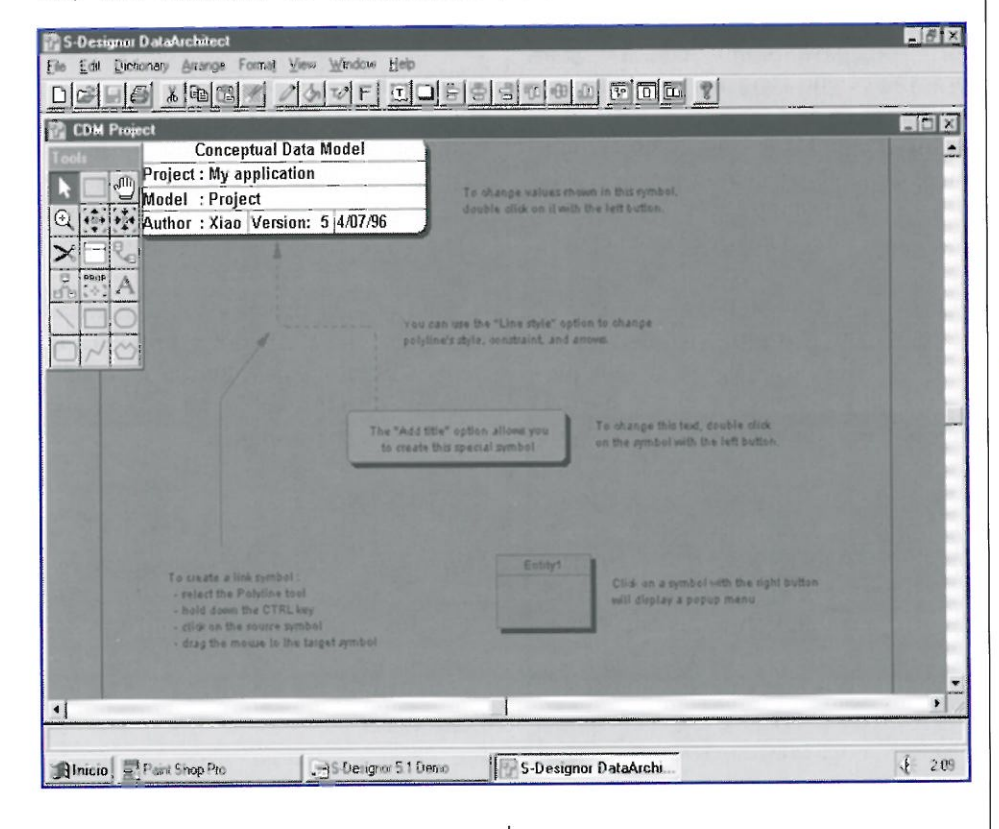
CONTENIDO DEL CD-ROM

S-DESIGNOR, VISUAL BASIC CCE, UTILIDADES SHAREWARE Y GUÍA-SP

En el CD-ROM correspondiente al presente número de SÓLO PROGRAMADORES se ha incluido una demo de S-Designor, la última herramienta RAD de desarrollo para PowerBuilder y Visual Basic de Sybase y la beta de la edición de creación de controles ActiveX Visual Basic 5.0 CCE, además de la GUÍA-SP, una selección de herramientas shareware de programación y los correspondientes fuentes de los artículos de la revista.

VISUAL BASIC 5.0 CONTROL CREATION EDITION

En el directorio \VBCCE se encuentra la beta de Visual Basic 5.0 Control Creation Edition, o lo que es lo mismo, la edición de creación de controles ActiveX de Visual Basic. Estos componentes software compactos y reutilizables se pueden integrar en una gran gama de productos que incluye Microsoft Internet Explorer, Office 97, el sistema de desarrollo Visual C++ y el sistema de gestión de datos Visual FoxPro.



La tecnología ActiveX permite crear de forma rápida aplicaciones basadas en componentes orientadas a Internet, intranet y entornos cliente/servidor.

Para instalar esta beta se deberá ejecutar el fichero VBCCEIN.EXE de este directorio, tras lo cual el programa preguntará si se desea instalar la beta. Tras pulsar en *Sí* aparecerá una pantalla con información acerca de la licencia, donde deberemos pulsar en el botón *Yes*. Una vez hecho esto, comenzará la instalación, que sólo se verá interrumpida para pedirnos el nombre del directorio de destino, que se puede cambiar mediante el botón *Browse* (en caso de no existir, se nos preguntará si deseamos crearlo automáticamente). Por último, se creará el correspondiente grupo de programas con sus respectivos iconos.

Para acceder al programa una vez instalado, tan sólo habrá que pinchar en el icono *Visual Basic 5.0 CCE*, que se encuentra en el grupo de programas del mismo nombre.

S-DESIGNOR 5.1

El directorio \SDESIGN contiene la versión demo de S-Designor 5.1, una herramienta RAD de Sybase que permite generar aplicaciones para PowerBuilder y Visual Basic de una forma rápida y sencilla.

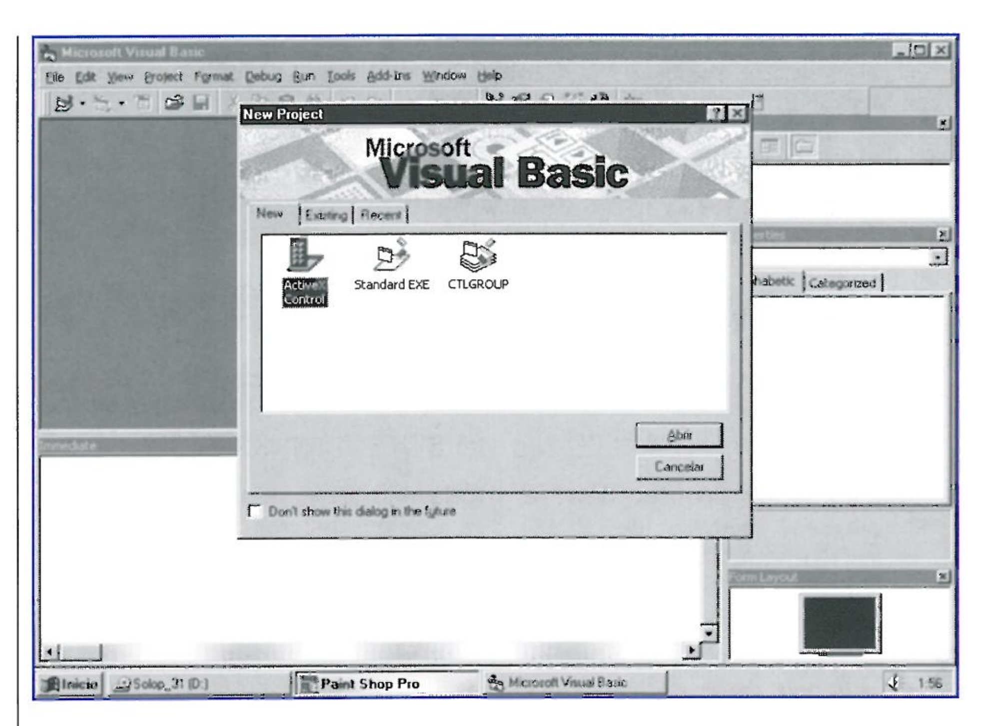
Para instalar cualquiera de sus componentes habrá que ejecutar en este directorio el fichero SETUP.EXE. Una vez hecho esto, aparecerá una ventana en la que habrá que elegir el producto que se desea instalar, que puede ser el visualizador de documentación *on-line* o los componentes *Data Architect*, *AppModeler* para PowerBuilder, *AppModeler* para Visual Basic, *Procces Analyst* o los drivers ODBC para Windows 95 o Windows NT, todos estos componentes para 32 bits. Sin embargo, si se pulsa en el recuadro *Display 16-bit programs* apa-

recerán en la lista, además, los mismos programas en su versión de 16 bits.

Una vez elegido el producto que se quiere instalar y pinchando después en el botón Install, comenzará el proceso de instalación. Lo primero que aparece en una ventana con los términos de la licencia, que desaparecerá al pulsar en Yes. Un nuevo cuadro nos hará decidir si se desea copiar los ficheros y/o utilizar nombres largos en Windows 95/NT. Tras seleccionar las opciones deseadas y pulsar en Next aparecerá una nueva ventana, donde se elegirá el directorio, modificable a través del botón Browse.

Una vez realizados estos pasos comenzará el proceso de instalación del programa, tras el cual se creará el grupo de programas correspondiente y sus iconos. Por último, para entrar al programa se deberá pinchar sobre el icono correspondiente a la aplicación instalada (por ejemplo, en el caso de Data Architect se pulsará el icono con ese mismo nombre).

NOTA: El espacio necesario en disco duro varía según el número de componentes que se instale. Obviamente, si sólo se instala uno de ellos hará falta disponer de menos espacio que si se instalan dos o más.



UTILIDADES

En el directorio \UTILS se incluyen una serie de herramientas y utilidades de programación visual para Windows, organizadas en distintos subdirectorios con el nombre de la herramienta en cuestión. Algunas de ellas tienen un fichero de instalación denominado SETUP.EXE. Otras, simplemente, vienen ya de por sí descomprimidas. Por último, algunas de estas utilidades están comprimidas en formato ZIP, por lo que se deberá utilizar el descompresor PKUNZIP para descomprimirlas. Este descompresor está incluido en el directorio \PKUNZIP.

GUÍA-SP

El directorio \GUIA-SP contiene la habitual guía de currículos de informáticos destinada a mostrar a las empresas españolas la demanda existente entre nuestros lectores-programadores.

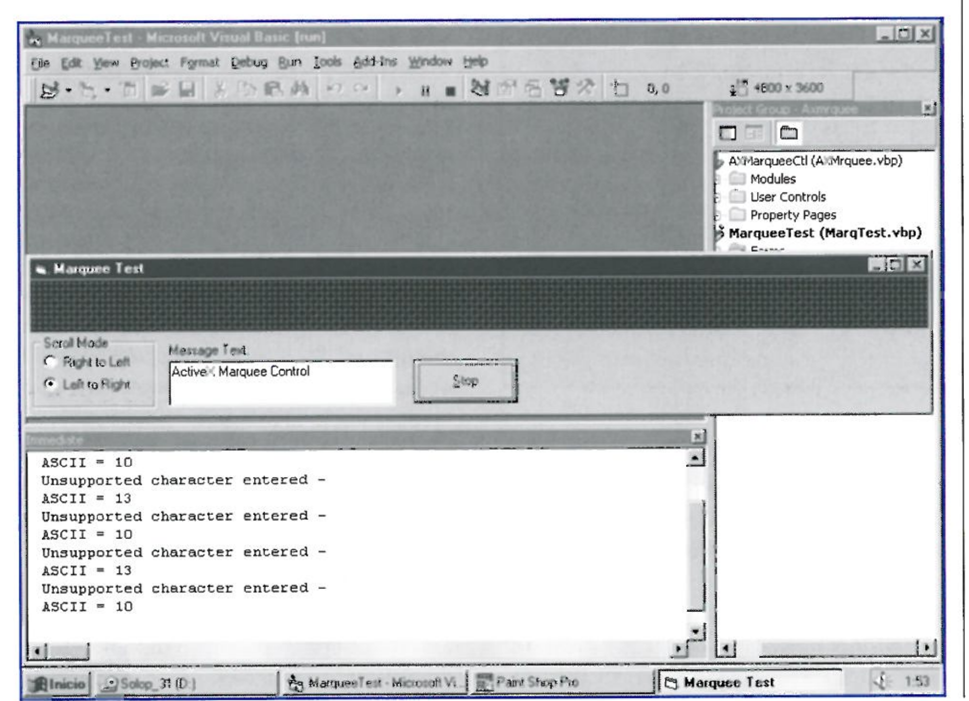
Esta guía se puede visualizar desde cualquier navegador consultando el fichero INDICE.HTM. Desde aquí animamos a todos los lectores a seguir enviando sus currículos.

BROWSERS

En el directorio \BROWSERS se encuentran varios browsers para navegar por Internet, con los cuales se podrá consultar también la Guía SP de Informáticos.

FUENTES

Por último, y como viene ya siendo habitual, en el directorio \DISK31 del CD-ROM se encuentran los archivos correspondientes a los artículos de la revista.



CORREO DEL LECTOR

En esta sección, los lectores de SÓLO PROGRAMADORES tienen la oportunidad de hallar respuesta a los problemas que puedan tener en cualquier tema relacionado con la programación.

ARCHIVOS PCX

P *Necesito hacer un programa que sea capaz de leer, procesar y crear imágenes almacenadas con el formato PCX. ¿Serían tan amables de proporcionarme información sobre dicho formato? Les quedaría muy agradecido.*

Juan Luis López Barroso (Murcia)

R El formato de los archivos PCX es algo más complicado de lo que puede parecer en un principio. Fue creado hace varios años y, la verdad, esto es algo que se nota nada más comenzar a trabajar con el mismo.

Todo archivo PCX comienza con una cabecera que proporciona la información necesaria sobre la imagen que contiene. Tiene un tamaño fijo de 128 bytes, lo cual representa una ventaja en muchas ocasiones. Si siempre utiliza archivos de la misma resolución y nœmero de colores (como puede ser su caso), se puede ignorar por completo. Pero siempre es mejor escribir un código genérico que sea capaz de trabajar con cualquier imagen almacenada con este formato.

La cabecera está compuesta por los siguientes campos:

- Fabricante (byte 0): Es un valor de 1 byte que especifica el fabricante. Su valor siempre suele ser 10 (archivo PCX de *Zsoft*).
- Versión (byte 1): Versión del formato utilizado en el archivo. También ocupa 1 byte.
- Compresión (byte 2): Se trata de 1 byte que indica si se ha utilizado compresión o no. En el primer caso, el byte es igual a 1.
- Bits por *pixel* (byte 3): Contiene el número de bits necesarios para representar un *pixel* de la imagen, por cada uno de los planos. Es un solo byte que puede contener los valores 1, 2, 4, o bien, 8.
- Ventana (byte 4): Son las dimensiones de la imagen. En primer lugar se indican las coordenadas mínimas (X e Y). A continuación aparecen las coordenadas máximas (en el mismo orden, es decir, la coordenada X seguida de la Y). En total son 8 bytes.
- HDPI (byte 12): Resolución horizontal de la imagen en puntos por pulgada (DPIs). Se utiliza para almacenar la resolución que fue utilizada a la hora de escanear la imagen. 2 bytes.
- VDPI (byte 14): Igual que el anterior, pero referido a la resolución vertical de la imagen. 2 bytes.
- Mapa de color (byte 16): Contiene la paleta para los modos de 16 colores. Está formada por tríos de valores, uno para el rojo, otro para el verde y el último para el azul. Cuidadó porque estos valores no son los que hay que poner en la paleta de la tarjeta de vídeo. Cada valor de color va de 0 a 255 y la paleta de la tarjeta sólo admite de 0 a 64, así que no hay más remedio que dividir estos valores entre 4 para interpretar la imagen correctamente. Este campo ocupa 48 bytes (16 colores x 3 bytes).
- Reservado (byte 64): Es un byte reservado que normalmente tiene asignado un 0.
- Número de planos (byte 65): Número de planos de color utilizados en la imagen. 1 byte.
- Bytes por línea (byte 66): Indica el número de bytes que es necesario reservar para interpretar un plano de una línea de la imagen. El nœmero no tiene por qué coincidir con la diferencia entre la coordenada X máxima y la coordenada X mínima. El valor siempre tiene que ser par. 2 bytes.
- Información de la paleta (byte 68): Sirve para saber cómo interpretar la paleta. Si vale 1, indica paleta de color o blanco y negro. Si es 2, indica paleta de escala de grises. 2 bytes.
- Tamaño horizontal (byte 70): Tamaño horizontal de la pantalla en *pixels*. 2 bytes.
- Tamaño vertical (byte 72): Tamaño vertical de la pantalla en *pixels*. 2 bytes.
- Relleno (byte 74): Se utiliza para completar el tamaño de la cabecera. Son 54 bytes que normalmente están puestos a 0.

Posiblemente se esté preguntando dónde se encuentra la paleta de las imágenes de 256 o más colores. Para las primeras, la paleta se almacena justo al final del fichero. Se utilizan, al igual que la paleta de 16 colores, tríos de valores correspondientes al rojo, verde y azul, por ese orden. Por tanto, son 768 bytes que, además, van precedidos de un 12 decimal para separar los datos de la paleta de colores. La existencia de esta paleta se indica mediante el byte de la versión del archivo, que contendrá el valor 5. En el segundo caso, las imágenes no contienen paleta. Simplemente se almacenan por líneas de rojo, verde y

azul. Como puede verse, es muy importante saber interpretar correctamente la cabecera para saber el tipo de imagen que contiene el archivo.

A continuación de la cabecera van los datos de la imagen. Estos datos se almacenan por líneas y, en caso necesario, también por planos. Por ejemplo, una imagen almacenada por planos se almacena de la siguiente forma: línea 0 del plano rojo, línea 0 del plano verde, línea 0 del plano azul, línea 0 del plano de intensidad, línea 1 del plano rojo, etc. Los datos suelen estar comprimidos con una técnica muy sencilla denominada "*run length encoding*". Para interpretarlo se utiliza este algoritmo:

```
PARA cada byte (X) del archivo
  SI los dos bits de mayor peso de X
están a 1 ENTONCES
    contador = los 6 bits de menor peso
de X
    dato = el byte siguiente a X
  SI NO
    contador = 1
    dato = X
  FIN SI
FIN PARA
```

Para leer un archivo PCX, en primer lugar hay que calcular sus dimensiones, calcular cuántos bytes se necesitan para leer cada línea (Número de planos * Bytes por línea) y después interpretar las líneas. La paleta de colores (si la hay) puede ser leída antes o después de los datos.

PROBLEMAS CON LA RECURSIVIDAD

P Hola, amigos. Llevo muy poco tiempo programando y tengo algunos problemas derivados, seguramente, de mi poca experiencia. La verdad es que no encuentro tan complicado el tema de la programación, pero sigo encontrándome con algunas dificultades. Mi problema es el siguiente: en un libro de programación en Pascal he leído una pequeña sección dedicada a la recursividad. Entiendo bien el concepto (es muy sencillo), pero soy incapaz de resolver un problema de programación con recursividad. No entiendo bien cómo funciona.

Armando Segura
(Madrid)

R La recursividad es un tema muy curioso. Unos piensan que es una maravilla y otros abominan de ella o no la entienden. Lo cierto es que resulta muy elegante utilizar una función recursiva para ciertas tareas. Por ejemplo, el método de ordenación *QuickSort* es recursivo por definición, y resulta más fácil y bonito implementarlo con una función recursiva que con una función iterativa. Así mismo, muchas funciones matemáticas se pueden implementar fácilmente mediante la recursividad. La parte negativa la proporciona la gran cantidad de memoria que necesita. Recordemos que una función recursiva no es más que una función que se llama a sí misma. El número de llamadas que se producen en la ejecución depende por completo del problema que se esté resolviendo pero, por lo general, siempre es muy alto. Esto significa que la pila del sistema se llena a gran velocidad porque en cada llamada se almacena información en la misma. Salvo en problemas sencillos, no merece la pena utilizar la recursividad.

Por fortuna, para los programadores con menos experiencia existe un truco infalible. A la hora de escribir una función recursiva hay que plantearse las siguientes preguntas:

- ¿El problema más sencillo se resuelve sin una llamada recursiva?

- ¿Las llamadas recursivas sirven para resolver un problema más pequeño que el original?

Por ejemplo, el siguiente programa calcula el factorial de un número:

```
Program FactorialRecursivo;
Uses Crt, Misc;
Function Factorial (Numero: longint):
longint;
begin
 if Numero = 0
   then Factorial := 1
   else Factorial := Numero * Factorial
(Numero - 1);
end;
var Num: longint;
begin
 clrscr;
 writeln (CALCULO DE LA FACTORIAL
DE FORMA RECURSIVA);
 writeln;
 repeat
   write (Introduzca un número entre 0 y
16:);
   readln (Num);
 until Num in [0..16];
 write (Num,! =, Factorial (Num));
 Pausa;
end.
```

La forma de calcular el factorial de un número N consiste en multiplicar N por el factorial de N-1, teniendo en cuenta que el factorial de 0 es igual a 1. Toda función recursiva tiene que tener un caso sencillo que sirve para realizar los retornos y evitar que la recursividad sea infinita.

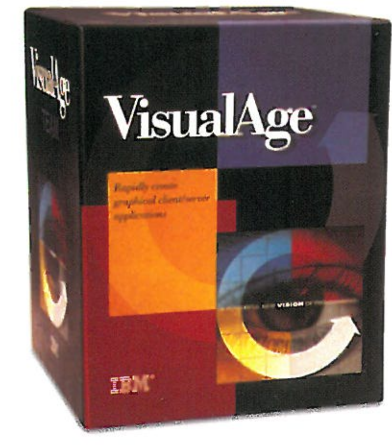
En este ejemplo, el caso más sencillo consiste en calcular el factorial de 0. Por tanto, la condición de salida mira si se está calculando el factorial de 0, en cuyo caso simplemente se devuelve un 1. Si no es así, se resuelve un problema más pequeño que el original. Siempre será más fácil calcular el factorial de un número pequeño que el de uno grande. En este caso, la llamada recursiva se hace para calcular el factorial de un número algo más pequeño que el pasado como parámetro. Según esto, quedan respondidas las dos preguntas y se observa que el esquema recursivo utilizado en el programa es bueno.

Tan sólo queda realizar un pequeño seguimiento de la lógica de la función recursiva (este caso es tan sencillo que casi no hace falta) para verificar que hace correctamente lo que tiene que hacer. Muchas veces las funciones recursivas no funcionan adecuadamente porque el programador no se ha planteado estas preguntas y se ha puesto directamente a programar sin reflexionar lo suficiente sobre el problema que le ocupa.



Foro VisualAge de IBM
900 100 400

Le invitamos a conocer VisualAge. Su empresa se lo agradecerá.



En la familia VisualAge puede encontrar: Java, C++, SmallTalk, COBOL, Basic, Generator, PACBASE y RPG.

Descubra cómo la familia VisualAge de IBM puede hacer que la información fluya fácilmente a través de su empresa y cómo opera en arquitecturas cliente/servidor multinivel. Analice las posibilidades únicas de VisualAge en desarrollo de aplicaciones con el lenguaje y sistema de su elección y en un mundo interconectado como el nuestro. Conozca casos prácticos de soluciones realizadas con VisualAge.

Le invitamos al Foro VisualAge de IBM:
• 11 de marzo en Madrid.
• 13 de marzo en Barcelona.
Para realizar su inscripción llámenos al 900 100 400 de lunes a viernes de 9 a 19h.
Visítenos en Internet: http://www.software.ibm.com/ad/visage



Soluciones para nuestro pequeño mundo